Введение в программирование как проектирование

Учебное пособие для школьников 7-10 классов
и их преподавателей

И.Р. Дединский

Кафедра информатики и вычислительной математики МФТИ

Кафедра системного программирования МФТИ

Институт системного программирования РАН

mail@ded32.ru, *vk.com/ded32_ru*, *t.me/ded32_ru*

Актуальная версия:

gg.gg/TXBook

storage.ded32.net.ru/Lib/TX/TXUpdate/TXBook.pdf

2024-04-09

Москва 2023

Введение в программирование как проектирование

И.Р. Дединский

Кафедра информатики и вычислительной *математики МФТИ*

Кафедра системного программирования МФТИ

Школа дает нам циркуль знаний для чертежа квадрата жизни.

Предисловие

Чем не является эта книга

Во-первых, эта книга не является учебником информатики. Информатика – сложная и большая наука, по ней уже есть хорошие учебники, используйте их. Это учебное пособие по проектному подходу в программировании. Оно для школьников, студентов и преподавателей, желающих сразу учиться профессиональному подходу, а не переучиваться впоследствии.

Во-вторых, это учебное пособие не претендует на полноту изложения, например, всего языка Си или тем более C++. Полнота в программировании вообще иллюзорна и слабо достижима, конкретные вещи, реализации, библиотеки, пакеты устаревают еще до того, как в типографии высохнет краска на первом тираже документации. Вам придется смотреть много дополнительной информации по другим книгам, гуглить в Интернете. По программированию, кстати, лучше искать на английском языке через Google, а не на русском через Яндекс – качество поиска будет гораздо выше.

Из учебников лучше всего книга Стивена Прата "Язык Си, лекции и упражнения" (последнее издание), ее легко найти в Сети. Она не про графику и не про проекты, зато там ясно и достаточно полно изложен сам язык программирования Си, который мы будем использовать.

Вместо детального описания языка программирования изложение здесь отталкивается от исходной задачи, которую должна решить программа, но задачи подбираются так, чтобы охватить принципиально важные вещи в разработке программ. Изложение фокусируется не на программистских трюках или особенностях конкретного языка, а на вещах, которые будут работать независимо от языков и платформ, как сейчас, так и в будущем. Причем так, чтобы не пришлось потом переучиваться при смене языков, библиотек, мест учебы и работы. Задача грамотного методического подхода – минимизировать это переучивание.

В-третьих, это пособие описывает не все языки программирования, а один – Си, где слегка используются незначительные элементы C++, в тех местах, где ограничения Си слишком мешают. Почему Си, а не Питон, Паскаль или Java, к примеру?

Главная задача первого языка программирования – дать максимальную свободу в последующем развитии, не замыкать в себе, не скрывать правду о том, как все устроено на самом деле, но и не быть навязчивым в этом. Языки, отталкивающиеся от математики, порождают "сферических программистов в вакууме", у которых все сложно с эффективностью кода. Языки слишком низкого уровня трудны для больших задач. Богатые языки с миллионом концепций привязывают к себе, потом сложно их поменять на что-то другое, ведь так жалко терять любимые "фишки" и приемы. Все это, разумеется, для начинающих, зрелый профессионал особых проблем не заметит. Но в начале пути к профессионализму процесс развития бывает довольно хрупок, и незачем его затруднять. Ну и не надо заменять обучение программированию обучением конкретному языку. Не надо привязывать и привязываться.

Си простой и ясный, если не лезть в тонкости (сложную указательную арифметику или нагромождения типов, к примеру – а это и не цель начального курса), быстро учится, отлично сбалансирован по краткости/эффективности, не скрывает реализацию, показывая связь с аппаратурой компьютера, имеет огромное число потомков, на которых потом легко перейти при необходимости, и поэтому хорошо подходит для обучения. В Паскале слишком много ограничений, воспринимающихся как неудобства, и люди "пускаются во все тяжкие", переходя на другие языки. Java хороша, но в ней слишком много "магии" и синтаксической нагрузки в начале, и она отчасти приучает к оверинжинирингу, то есть избыточной сложности и стоимости решений. Питон и JavaScript поначалу легки, но на них трудно отличить действительно эффективную конструкцию от неэффективной, и для этого надо знать неожиданно много в их реализации, и много неявного в коде. Кроме того, у них плохой баланс по эффективности – любимые всеми краткие записи совсем не означают действительно эффективных решений. У чистого C++ такие же беды, плюс в последнее время бурный рост в сторону синтаксической сложности. Ассемблер имеет большой входной порог. Другие языки пока менее распространены, и создают большие трудности при переходе с них. Разумеется, это все не про реализацию конкретных индустриальных задач профессионалами. Это про обучение. Но для начинающих первый шаг по сложности заложенных в него процессов сравним с проблемами, стоящими перед опытными разработчиками. И да, мы не минимизируем усилия учителя, беря язык, с которым ему легко. Мы максимизируем пользу для обучаемых.

В-четвертых, для того чтобы рассмотреть всерьез какой-либо аспект или тему, в книге часто предлагается предварительная задача, в которой вы должны реализовать нужный функционал без каких-либо новых знаний, на грани "как-нибудь", и немного помучаться при этом. Новое часто появляется и лучше воспринимается как результат анализа трудностей при реализации старыми, традиционными способами. Поэтому не бросайтесь сразу читать наперед "как надо делать". Если предлагается реализовать что-то предварительно, сделайте это. Последующий анализ трудностей и переделка (рефакторинг) заложит лучшее понимание и правильные рефлексы профессионального программиста, которые помогут вам работать грамотно вплоть до пенсии. Если же бежать за спойлерами, что ж – код вы напишете, а нужные привычки к вам не придут. Вы получите пустышку. Но обратная сторона этого – хорошие идеи здесь не даются сразу, в готовом виде, и на их освоение требуется больше времени.

Если где-то явно не сказано, что вот перед вами обязательное задание, а просто есть предложение какой-то идеи, то все равно постарайтесь ее реализовать. Программируйте больше, делайте разные варианты, сравнивайте их. Ничто не заменит практику, особенно с последующим ее обдумыванием и обсуждением.

В обсуждении, с кем угодно, невзирая на авторитеты, если вам предлагают лишь единственное решение – считайте, что вас слегка обманули. Решений всегда много, и очень полезно их придумывать и обсуждать, даже самые неудачные. Бывает так, что неудачное в одном случае решение становится очень выигрышным при смене условий. А смена условий задачи – неотъемлемая часть реальной жизни профессионала, к чему он должен быть всегда готов (это даже имеет специальное название – масштабирование).

В-пятых, автор достаточно суров к читателю. Он не сюсюкает, не повторяет одно и то же по нескольку раз, как это принято в учебниках. Он считает, что если сказано – то сказано, мы договорились делать так-то и так-то, и если читатель это принял на этапе чтения книги – он это делает везде в коде. Везде. ВЕЗДЕ. Не надо оправдываться, что в таком-то куске кода вам неохота было форматировать строки ровно. Или неохота было называть переменные по-нормальному, и вы решили назвать по-быстрому, покороче. Или еще что. Нет. Код будет кривой, и виноват в этом не автор. В задачу не входило раздуть книгу до двух тысяч страниц

повторами одного и того же. Вместо этого перечитайте раздел десять раз сами. А потом подумайте, почему это написано и почему именно так написано. Достройте мысль автора до своего проекта или кода. Простым повторением и работой "лишь бы работало" это не осилить. Автор не собирается писать или думать за читателя. Да, так. Без сюсюканья и подтирания соплей, сорри. Это профессия, а не развлекуха.

Конечно, автор понимает, что даже при очень большом желании научиться хорошо – это "хорошо" может быть разным, и поэтому не надо тильтовать от абзаца выше. Метод, там описанный, к сожалению, будет работать только в идеальных условиях, которые редко встречаются. В реальности вам будет нужна обратная связь, мнения, критика буквально по каждому вашему шагу, по каждой функции и строке кода. Не потому, что вы неспособны их написать. А потому, что способов написать – огромная масса, из них хороших способов – единицы. И как вам вырулить в вашей личной образовательной траектории без грамотного заинтересованного преподавателя, ментора, знакомого с методикой проектного обучения – автору, увы, непонятно. Это важнейшая причина, почему эта книга не писалась так долго. Автор не уверен, что она обязательно сработает во всех случаях, для всех учеников, и виноват в этом не он и не читатель, а сам формат неинтерактивного общения, что с книгой, что с видеороликами. Нужен живой человек, активно взаимодействующий с обучаемым, и никому до сих пор не понятно, как его заменить в общем случае, чтобы метод срабатывал всегда и для всех.

Вообще, один из главных трудов по информатике называется "Искусство программирования". Великий Дональд Кнут знал, как выбирать название. Искусство.

Но будем надеяться, что эта книга, хоть и не панацея, но +1 шанс вырасти как программист правильно. Будем надеяться, что он сработает с тобой, читатель.

Наконец, в-шестых и в-главных. Автор не ставил своей целью написать "еще одну книгу по Си". Или какому-либо другому конкретному языку программирования. Или "по алгоритмам". Или "по информатике". Таких книжек достаточно. Автор хотел показать то, что отсутствует в многих книгах "по чему-то", но неизбежно необходимо каждый день в профессии разработчика. Эта книжка про программирование как конструирование, программирование как проектирование, про основы этого подхода, и это составляет суть обучения будущих профессионалов. Размер имеет значение, и основная сложность современного программирования – разработка больших систем и работа с большим объемом кода, от двух-трех тысяч строк. Профессионал, работая с сотнями тысяч строк, на эти цифры только посмеется, потому что он воспитан в духе системного подхода, а начинающий без нужных навыков, превратившихся в рефлексы, будет писать такую программу всю жизнь – одно починишь, другое сломается. Поэтому без навыков работы с большим кодом будет тупик. Нельзя, щелкая только алгоритмические задачки, потом легко писать большие проекты. Придется трудно переучиваться. Но зачем, когда можно начать уже с более-менее больших задач, только логически простых. А затем добавлять логическую и алгоритмическую сложность.

Конечно, здесь приведено не все и сказано не все, но это книжка про первый шаг. Да и кто из начинающих сейчас будет читать фолиант в тысячу страниц? Если кто-то будет – смело берите Стива МакКоннела, "Совершенный код". Там все есть. Ну почти все. :) Но она рассчитана на уже достаточно знающих людей, для которых многое написанное там очевидно. Такая вот загадка, чтобы стать профессионалом, надо сначала стать профессионалом. Поэтому здесь предлагается пропедевтический подход – уровень учебных проектов большого (для начинающих), но не запредельного объема, сохраняющих многие свойства реальных задач,

плюс техническое воображение, масштабирующее учебный проект до сотен тысяч строк реального кода.

Последнее, можно ли научиться только по этой книжке, без хорошего преподавателя с опытом профессиональной разработки, не экономящего свое время и силы на учебном процессе и согласного "влезать в голову" каждого ученика? Таких еще поискать, кстати. Увы, тут автор не уверен, что это точно сработает. Без обратной связи по каждой строчке кода, выверенной с учетом темпа и особенностей индивидуального развития каждого ученика, этого не достигнуть. Либо ученику надо быть отъявленным перфекционистом, и не сломаться при этом и не бросить. Непростая задача.

Однажды один из студентов автора задал вопрос – младший брат двенадцати лет хочет заниматься программированием, как ему учиться наверняка, на какие курсы идти и какие книжки читать, чтобы точно выучиться хорошо, без перекосов и последующего переучивания? Трудный вопрос, и ответ получился такой – писать любые программы на любом языке, но объемом не менее 7000 строк собственного, не заимствованного кода. Либо научится, либо бросит. Может быть, это эффективно, но несколько жестоко, и эта книжка – некоторая попытка внести в этот процесс гуманности. Брат студента, кстати, научился, поступил на Физтех, успешно там учится и не жалеет. Чего и тебе желаем, читатель.

Профессионально-ориентированный подход к довузовскому преподаванию программирования (предисловие для преподавателей)

IT-индустрия в настоящее время является одной из глобальных системообразующих отраслей, ее развитие во многом определяет технологическое развитие других областей. Характерной чертой IT-индустрии является сильная зависимость от уровня компетенций специалистов (человеческого фактора), которая, в свою очередь, зависит от уровня профессионального образования. Высокая скорость развития информационных технологий, прежде всего программирования и разработки ПО, делает крайне проблематичным серьезное изучение профессии начиная лишь с ВУЗа – зачастую от тех, кто начинал еще со школы, наивный первокурсник уже отстал навсегда. Поэтому для успешного состояния IT-отрасли критично необходимо развивать образование довузовское, как это происходит в областях, близких к программированию – математике и физике, и где старт углубленного обучения по факту происходит уже в 6-8 классе школы.

Российское довузовское образование в настоящее время содержит некоторую возможность углубления в области информационных технологий. Если оно связано с углубленной математикой и физикой (а не предполагает лишь офисные технологии), то оно может приносить плоды и для развития IT-индустрии. Однако нынешние тенденции такого образования в России на данный момент расходятся как с современными требованиями к компетенциям IT-специалистов, так и с образовательными подходами за рубежом, что порождает странную на первый взгляд картину, когда в соревнованиях по спортивному программированию участники из России часто занимают весьма высокие позиции, а в то же время профессиональное сообщество российских программистов испытывает сильный недостаток высококвалифицированных кадров. В результате мы повсеместно пользуемся импортированным в Россию программным обеспечением и технологиями, что само по себе не есть плохо, но не добавляет российскому технологическому пространству ни развития, ни стабильности. Российские IT-специалисты, а вслед за ними и компании, в массе своей конкурентно проигрывают зарубежным по уровню компетенций, но совсем не потому, что имеют худшие способности. Просто их «так учили».

Углубленный подход к преподаванию информатики в большинстве случаев применяется в учебных заведениях выраженной физико-математической направленности и предполагает курс программирования, что безусловно правильно. Проблема в том, что в большинстве случаев способом реализации курса является решение большого количества изолированных алгоритмических задач (так называемый «олимпиадный подход», часто называемый «алгоритмическим»).

Однако, если ограничиваться только таким подходом и игнорировать современные тенденции развития IT-отрасли, зачастую получается, что даже успешный олимпиадник испытывает серьезные проблемы с успешностью при попытках выйти за пределы олимпиадной стилистики, например, при вступлении в профессию.

Существующее доминирование чисто спортивных тенденций практически не учитывает тот факт, что участие в разработке ПО, как для научных целей, так и в качестве инженерной профессии, – процесс проектно-ориентированный, а это требует многих качеств, которые в олимпиадном (алгоритмическом) подходе не нужны и, как следствие, не развиваются. Безусловно, владение алгоритмическими основами обязательно для будущего профессионала, однако его опыт никак не должен этим исчерпываться (см. рис. 1).

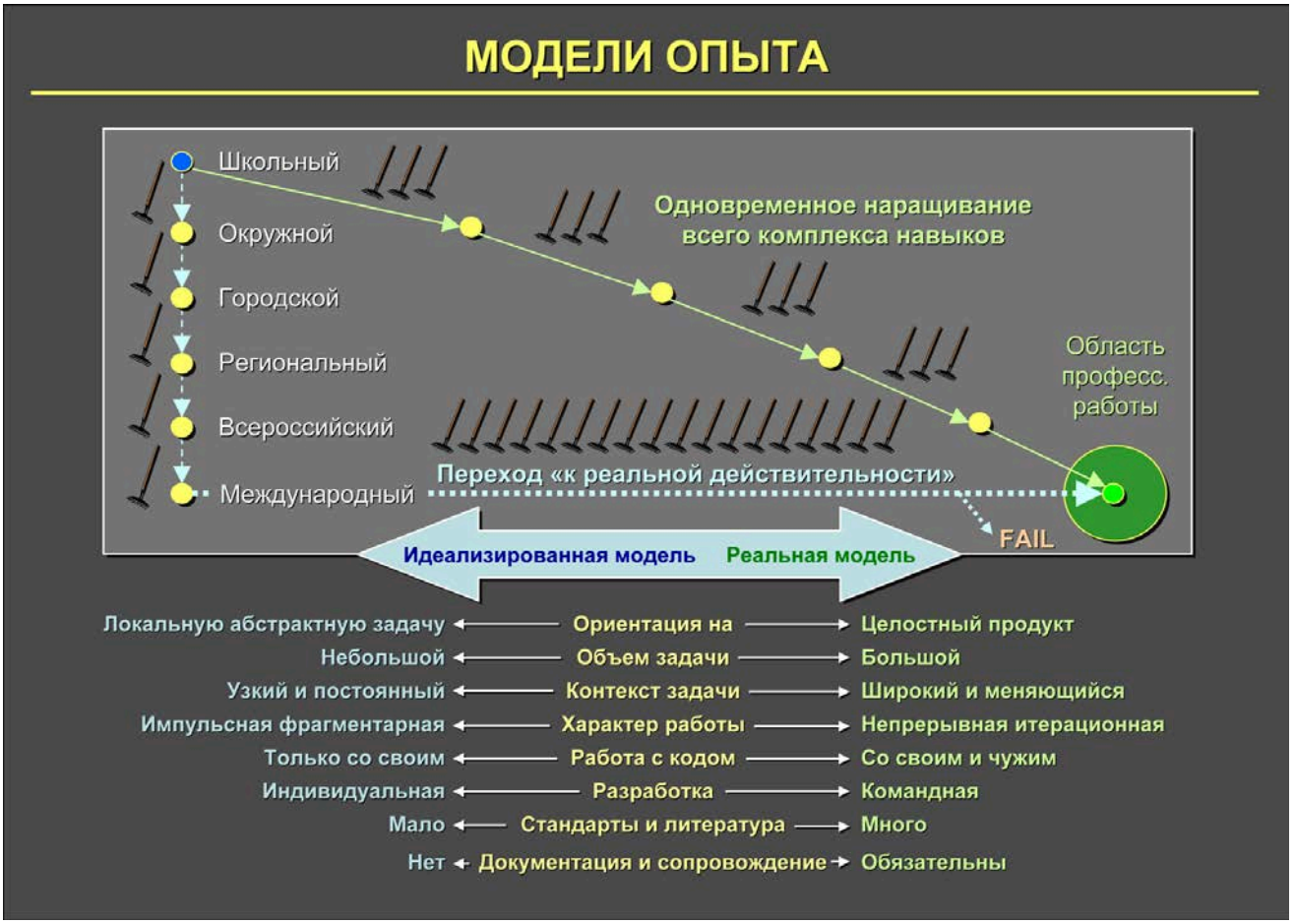


Рис. 1. Модели опыта и особенности динамики его набора в проектном и олимпиадном подходах.

«Граблями» обозначены трудности, в ходе которых набирается необходимый опыт. См. [6].

Олимпиадный подход, вследствие формально-соревновательного, зачастую спортивного характера, доминируя сейчас в российском образовательном пространстве, преследует в основном внутренние цели, необходимые лишь для реализации предельно краткого жизненного цикла решения (написал – сдал – забыл), и которые лишь частично коррелируют с профессиональными компетенциями, необходимыми для длительного жизненного цикла

разработки ПО. Иногда эти цели даже противоречат таким компетенциям (например, умению конструировать грамотный инженерно-технический компромисс), что дает описанный выше диссонанс между спортивными успехами и плачевным состоянием дел в российский индустрии. Полностью олимпиадный подход соответствует только одной профессии – олимпиадного тренера (или организатора олимпиад), что делает эту систему замкнутой.

В то же время профессионально-ориентированный (проектный) подхода совсем не противоречит олимпиадному движению, так проектный подход включает и концепции автоматического тестирования, которые приняты за основу олимпиадных технологий в информатике. Фактически, алгоритмический подход – основа олимпиад – это одна из частей проектного подхода. Однако на данный момент подавляющее доминирование именно спортивной составляющей мешает развитию профессиональной ориентированности, так что дело не в антагонизме, а в восстановлении паритета.

В результате характерный для каждой профессии диссонанс между «тем, чему учили» и «тем, что необходимо в профессии», описывается весьма непустым множеством образовательных разрывов, которые в настоящее время учащийся и студент должен преодолевать сам, и которые составляют его личный опыт. Такая ситуация существует и в школе, и в ВУЗе уже очень долго. В то же время, большинство разрывов типичны и легко обнаруживаются в ходе внимательного анализа (см. рис. 2).

Рис. 2. Образовательные разрывы между источниками знаний. См. [6].

Цель настоящего подхода – проанализировать образовательные разрывы и построить курс таким образом, чтобы минимизировать эти разрывы и максимизировать набор конструктивного положительного опыта, не ограничивающимся лишь конкретными приемами, шаблонами и

средствами. Это позволяет учащимся в дальнейшем ориентироваться в динамичном мире информационных технологий, которые часто успевают развиться и умереть до того, как по ним выйдет первый учебник. В таких условиях главная учебная задача, и не только в сфере ИТ, – научить будущего студента действовать грамотно и самостоятельно. Под грамотностью здесь понимается умение классифицировать проблемы, знать типовые решения, выбирать из них спектр адекватных решений, комбинировать их, придумывать новые решения, контролировать качество, мыслить не рецептами, а как минимум технологиями.

Для этого автором вводится понятие когнитивно-технологической единицы (КТЕ), как единицы действительного усвоения знаний, определенной следующим образом [1]:

1. зачем это надо,
2. что это такое,
3. на чем основано и с чем связано,
4. как это применять,
5. где это можно и где нельзя использовать,
6. чем придется пожертвовать,
7. что будет, если этого не делать,
8. какие в этом «подводные камни» (чего опасаться при применении).

Разработанный курс рассчитан на математически сильных и высоко работоспособных учащихся 7(8) – 10(11) классов физико-математического профиля, нагрузку 4-6 профильных учебных часа в неделю, развитую систему факультативов и консультаций. В преподавании используются принципы, отличающиеся от алгоритмического (олимпиадного) подхода (см. табл. 1):

Табл. 1. Сравнение особенностей проектного и олимпиадного (алгоритмического) подходов.

	Проектный подход	**Олимпиадный (алгоритмический) подход**
1.	Главными методологическим принципом является системный подход.	Главными принципом является соревновательный подход с судейством по формальным правилам.
2.	Во главу угла ставится задача, понимаемая как часть проекта, и, главное, путь от задачи к решению, а не кодирование алгоритма. Полученные решения повторно используются в дальнейшем, как и в реальной индустрии, что заставляет их реализовывать более системно и внимательно, увязывать с другими решениями. Работа основана на реальном (длительном) жизненном цикле разработки ПО.	Рассматриваются изолированные задачи, общие лишь по алгоритмической тематике. Использование реализаций полученных решений запрещено в последующих задачах. Сборки решений в цельный проект не предусматривается. Жизненный цикл решения предельно краток (написал – сдал – забыл).
3.	Проектный подход включает в себя тестирование реализованных алгоритмов (т.е. того, на чем основан олимпиадный подход), что соответствует положению дел в индустрии.	Олимпиадный подход не включает проектный и зачастую пытается рассматривать его как ненужного конкурента.
4.	Технологичность, надежность и масштабируемость решения ставятся выше «программистских трюков», иногда позволяющих в отдельных случаях добиться несколько лучших результатов. Активно	Если для задачи допустимо неуниверсальное, немасштабируемое, «неинженерное» (и иногда по сути неверное), но оно формально допустимо, то оно все равно годится для соответствия конкретным условиям

	Проектный подход	**Олимпиадный (алгоритмический) подход**
	рассматриваются альтернативные решения, полезные при масштабировании проекта.	соревнования. Рассмотрение альтернативных решений в этом виде образовательного процесса вторично.
5.	Понимание и корректное использование учащимся тех средств, с помощью которых он решил задачу, ставится выше уровня самих средств решения. Выполняется рецензирование исходного текста преподавателем, сдача работы производится в форме собеседования. Подход к оцениванию – комплексный.	Исходный текст программы и технологичность решения не оценивается (его смотрят только при подозрении на прямой плагиат), тестирование ведется без участия преподавателя. Единственный критерий оценки – количество автоматических тестов, пройденных программой. Собеседование по итогам решения не влияет на оценку.
6.	Самостоятельность решения является ключевым условием, которое необходимо доказать при сдаче работы.	Системы тестирования решения способны детектировать прямой плагиат, но заимствование с частичной переработкой детектировать не способны.
7.	Для записи алгоритма на языке программирования выбирается подмножество наиболее универсальных средств языка, чтобы не акцентировать внимания на кодировании и для более легкого перехода на другие языки программирования.	Активно используются особенности конкретных языков программирования, позволяющих легче решить определенные задачи, что «привязывает» учащихся к конкретным языкам и технологиям, которые завтра изменятся.
8.	Задачи ставятся в нескольких вариантах различной сложности (от базового до творческого), при сдаче работы засчитывается решение на любом уровне. Уровень сложности фиксируется и используется как дополнительная информация к оценке, для выяснения и повышения уровня профессионализма ученика.	Понятие творческого подхода не существует в силу предельной формализации критериев оценивания, однако существует деление задач по их сложности.
9.	В обучении активно применяются современные парные и групповые техники, использующиеся в индустрии (обмен кодом и документацией, перекрестное рецензирование и тестирование, групповая разработка стандартов взаимодействия компонентов проекта). Эти же техники используются при подготовке к ЕГЭ по информатике.	Групповые техники формально существуют, но участники контеста делятся либо по задачам (каждый решает свою), либо по «специализациям» – один знает больше алгоритмов, другой быстрее отлаживает программы и т.п. Такие техники не способствуют объединения учебных навыков у участников. Не развиваются технологии настоящей групповой работы.

Важнейшей задачей данного курса является формирование системы профессиональных ценностей (предпочтений) ученика (см. рис. 3). В конечном счете, это формирование и есть основная инвариантная методологическая задача курса, так как все остальное – лишь технология и будет неотвратимо изменяться с течением времени.

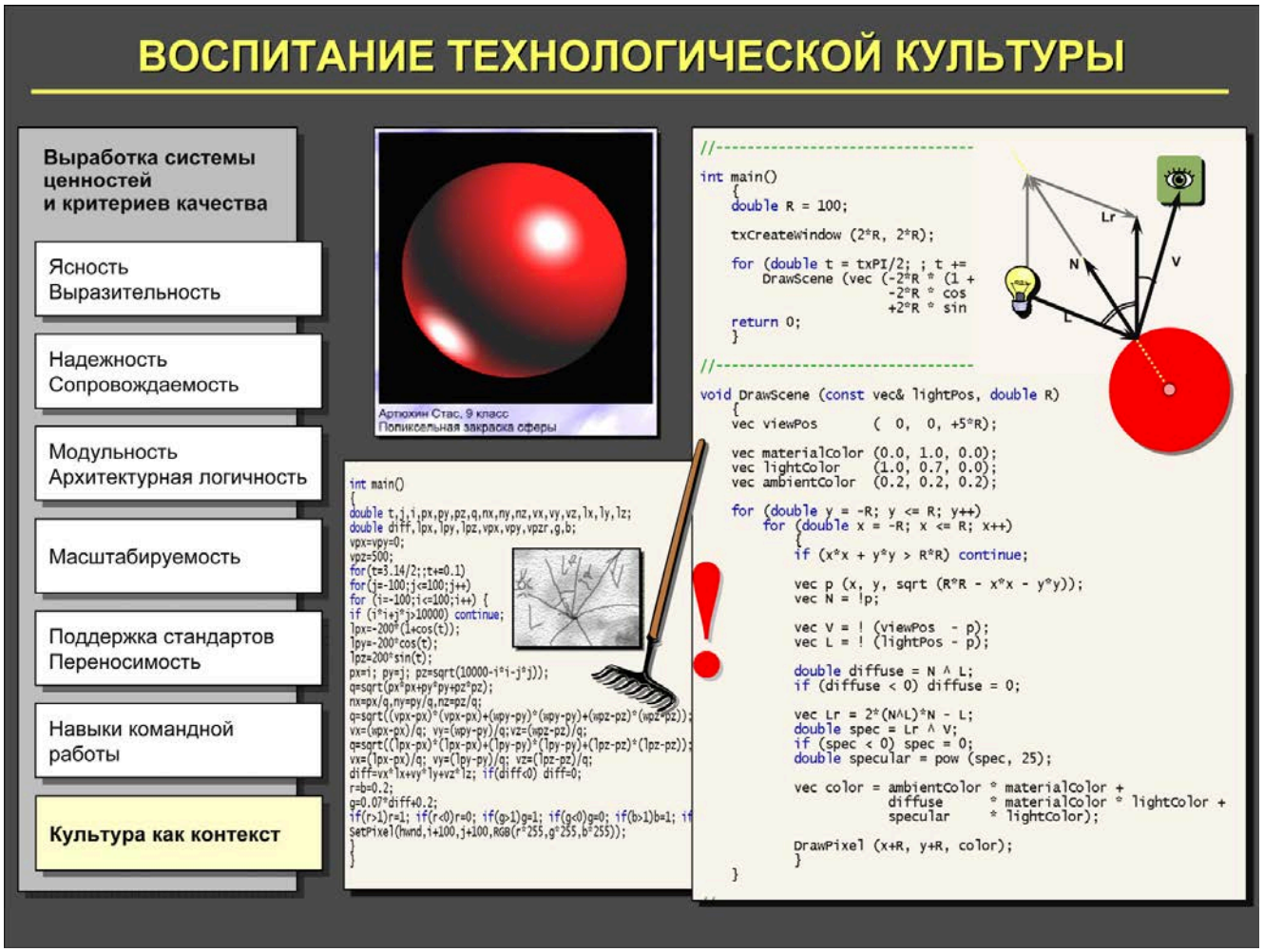


Рис. 3. Выработка системы профессиональных ценностей учащегося. В центре снизу показан фрагмент программы типичного студента, занимавшегося олимпиадным подходом. Справа представлен фрагмент программы девятиклассника, второй год обучавшегося по проектной методике. Алгоритмы программ идентичны (попиксельная закраска трехмерной сферы). См. [6].

Подход, ориентированный на проектную работу, сильно увлекает многих учеников и дает не только высокие проектные результаты (призовые места на Всероссийских и международных конкурсах – финале Intel Scientific and Engineering Fair (ISEF), Балтийском научно-инженерном конкурсе, конкурсах «Ученые будущего», «Интел-Юниор», «Интел-Династия-Авангард», «Старт в науку», «Шаг в науку»), но и высокие олимпиадные (победителей и призеров Всероссийского и международного уровня). Однако, так как он не совпадает с распространенным сейчас олимпиадным подходом, то он не всегда одобряется приверженцами олимпиад, которые зачастую хотят получить ученика «целиком и полностью», проявляя ярко выраженный монополизм. Они часто воспринимают проектную работу высокого уровня как конкуренцию, оттягивающую от них сильных детей, и пытаются создать для нее неблагоприятную среду, в том числе необъективными и не всегда корректными методами. Тем не менее, по-настоящему сильные олимпиадные школы России видят в проектно-профессиональном подходе большую перспективу (см., например, письма руководителей и сотрудников широко известных в программировании и прикладной математике университетов СПбГУ ИТМО [2, 3], МФТИ [4]). Зарубежные учителя информатики также признают этот подход крайне плодотворным [5].

Результатом прохождения курса становится не только понимание основных принципов программирования и владение основными алгоритмическими конструкциями, но и серьезные концептуальные и технологические навыки, позволяющие самостоятельно разрабатывать проекты достаточно большого для школьников объема (порядка курсовой работы 2-3 курса

ВУЗа, а иногда и бакалаврского диплома), успешно работать в групповых проектах, требующих активного взаимодействия участников, а некоторым – участвовать и регулярно побеждать в различных конкурсах и олимпиадах Всероссийского и международного уровней, участвовать в научных конференциях РАН наравне со взрослыми (рис. 4).

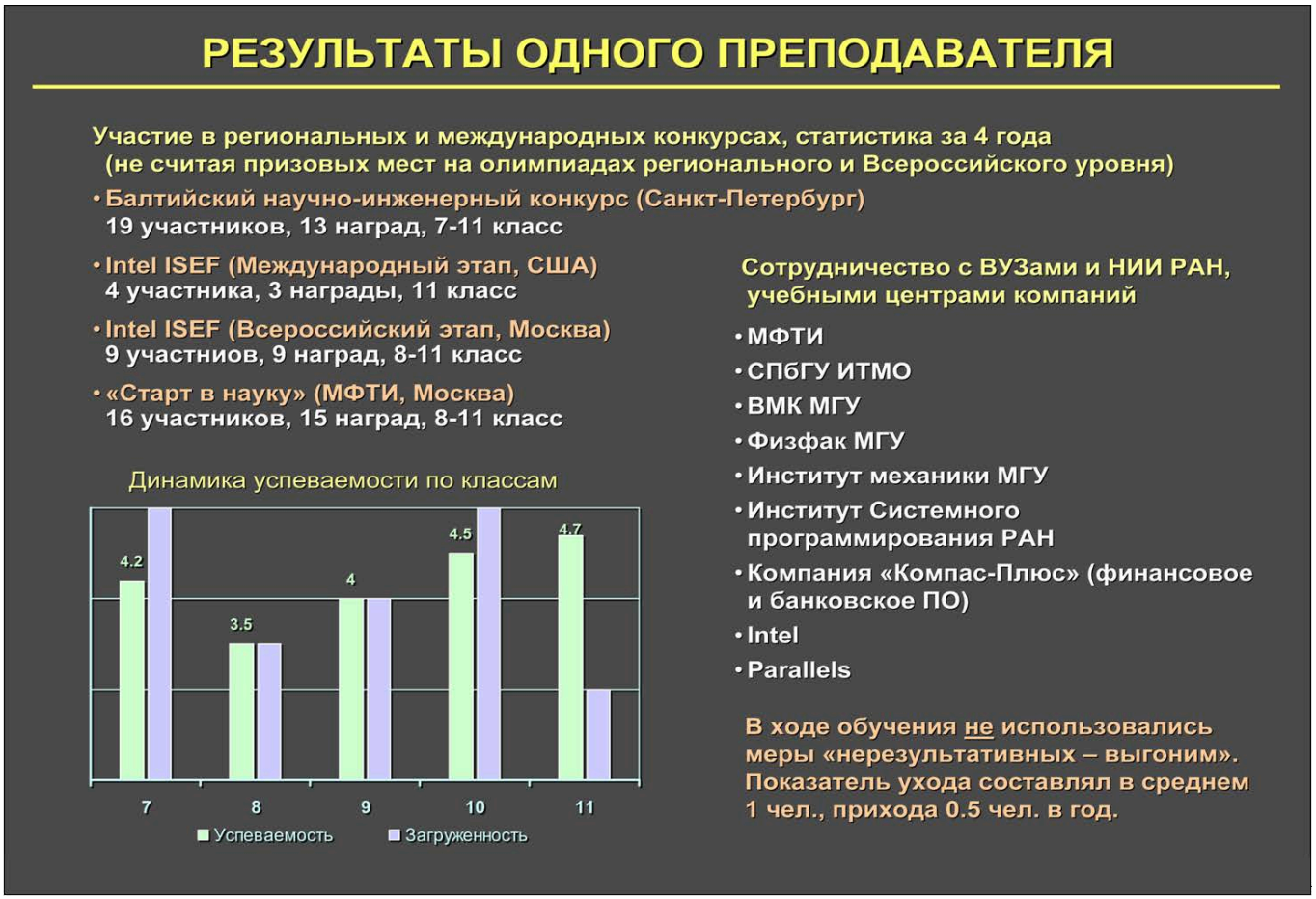


Рис. 4. Статистика результатов автора за время проведения педагогического эксперимента (2005–2009 г.г.). См. [6].

Свежий пример – в 2015 году на пилотной смене в образовательном центре «Сириус» (г. Сочи) проектные работы учеников были представлены В. В. Путину как главные научные достижения смены и были высоко оценены экспертами из правительства России, МФТИ и ИППИ РАН [7] (рис. 5).

Рис. 5. Проектные работы школьников представлены В. В. Путину
в образовательном центре «Сириус» (г. Сочи). См. [7].

К сожалению, на данный момент четко работающего методического «канала», который закономерно привел бы увлеченного школьника к профессии программиста, в России нет. Существующее дополнительное образование эту задачу не решает в силу его нерегулярности и частой методической несогласованности. Программисты в России растут «как трава», «как-то сами», и поэтому их так мало и из них еще меньше хороших. Представленный подход способен решить вопрос наличия такого «канала», с помощью которого можно получить большое количество высококлассных специалистов для развития российской индустрии информационных технологий.

Литература

1. И. Р. Дединский. Как хотеть учиться. // Компьютерра. – 2005, № 24 (тема номера: «Тьма просвещения»). https://old.computerra.ru/2005/596/207625/.
2. Проф. В. Н. Васильев, проф. В. Г. Парфенов, проф. А. А. Шалыто, А. С. Станкевич, Г. А. Корнеев (СПбГУ ИТМО). Письмо об оценке работы экспериментальной образовательной площадки. // 2009, http://ded32.net.ru/news/2009-04-04-32.
3. Проф. В. Г. Парфенов, проф. А. А. Шалыто (СПбГУ ИТМО). Письмо в защиту инновационной работы. // 2010, http://ded32.net.ru/news/2010-09-10-55.
4. Проф. А. А. Шананин, проф. И. Б. Петров (МФТИ). Оценка работы экспериментальной площадки. // 2009, http://ded32.net.ru/news/2009-09-01-39.
5. И. Р. Дединский. Курсы по методике преподавания программирования для учителей Южной Кореи. // 2009, http://ded32.net.ru/news/2009-08-20-37.
6. И. Р. Дединский. Аналитический подход к довузовскому преподаванию программирования – принципы преподавания и итоги эксперимента. Тезисы доклада и презентация. // Труды Всероссийского съезда учителей информатики в МГУ. – 2011, М., Изд-во МГУ. http://ded32.net.ru/news/2011-04-03-58.
7. И. Р. Дединский. Проектные работы по программированию представлены Президенту России В. В. Путину. // 2015, http://ded32.net.ru/news/2015-09-02-77.

Предупреждение, или TXLib – это всего лишь инструмент

В этой книжке разбираются в основном проекты, связанные с двумерной компьютерной графикой, близкой к разработке игр. Так как профессиональные инструменты для этого довольно сложны, автор разработал библиотеку TXLib, облегчающую разработку графических приложений под ОС Windows для начинающих. Она очень проста в использовании, но устроена далеко не идеально, чтобы показать проблемы работы с реальными программными библиотеками, и чтобы не привязать к себе – через некоторое время ее захочется сменить на что-то другое, и прекрасно. Только найдите перед этим все мемы и рофлы в документации и коде. Заодно в нем разберетесь, ахаха.

Почему графика? Точнее, почему сразу графика, почему начало не с алгоритмов? Дело в том, что алгоритм – лишь часть программы как системы, и, если он применен неправильно, слишком формально, не окружен нужным образом всеми подсистемами программы, то он будет бесполезен. Реальные проекты нередко состоят из сотен тысяч и миллионов строк кода и содержат многолетнюю работу многих сотен людей из разных стран. В результате код читается намного чаще, чем пишется. Требования к проекту часто сформулированы не так четко, как хочется, а вдобавок они еще и меняются – и проект должен к этому адаптироваться. В сочетании с общей сложностью работы такие проекты вызывают ощущение глубокой беспомощности у начинающих и у тех, чье обучение программирование происходило преимущественно на контестах и олимпиадах по программированию, ведь в олимпиадных задачах было все наоборот: четкие условия, малое количество соавторов, формальная проверка, небольшой объем кода. Профессиональное программирование – это не спорт, поэтому в нем присутствует такой огромный фокус на архитектуру, качество написания, масштабируемость, культуру. Поэтому людям с "контестным" (от слова "контест"), или олимпиадным стилем программирования приходится тяжело переучиваться, да ещё и не у всех получается. Знание алгоритмов – безусловно, важный навык, просто они не могут быть применены, если нет умения работать с реальными задачами так, чтобы более опытные коллеги не морщились на ваш код и ваши решения.

Графические программы и компьютерные игры с простыми сюжетами – хороший способ научиться программированию на реальных задачах, с правильными принципами, подходящими для профессиональной работы, и вам не придется кардинально переучиваться даже при смене языков и платформ. Они достаточно просты алгоритмически, но требуют достаточно большого объема кода, а умение работать с таким объемом и есть главная черта профессионала. Поэтому в предисловии и был упомянут объем кода в 7000+ строк. Самые первые проекты, конечно, будут меньше, но вы почувствуете, что код разрастается, и потребуется умение с ним управляться. Принцип тут один – разделяй и властвуй, но в программировании он слишком многогранен, чтобы остановиться только на этих словах. Приходится много думать, воображать, конструировать, чтобы реализовать игровой сюжет и при этом не запутаться в коде. Так что разработка игр – хорошая школа программирования. А TXLib несколько облегчает эту разработку.

Но есть одна проблемка. Библиотека TXLib – это всего лишь инструмент для того, чтобы облегчить первые шаги в программировании. Однако этот инструмент, как и любой другой, может быть применен неправильно. (Тем не менее, в основу TXLib заложены некоторые принципы, помогающие конструктивному неиллюзорному обучению.)

Сама по себе любая библиотека или язык программирования не научит начинающего писать программы грамотно. Научит этому разработка своих, достаточно больших проектов, в сочетании с тесным общением с профессионалами, желающими помочь начинающим. Такие

профессионалы должны обладать и опытом разработки больших программ, и педагогическими навыками, чтобы передать свой опыт начинающим. К сожалению, не всегда это совпадает. Профессионалы-программисты зачастую не хотят лезть в обучение, где хватает своих проблем. С другой стороны, иногда даже в сильных школах и курсах, хватаются за удобные инструменты обучения (чужие или свои библиотеки, среды и языки программирования), не следя за качеством кода обучаемых, за стилем и направлением их мышления[1], и тогда получается лишь видимость обучения. Такие образовательные иллюзии очень вредны. Заметны они становятся достаточно поздно, когда выясняется, что ученик, легко пишущий небольшие программы (пусть даже алгоритмически насыщенные, олимпиадные), принципиально не способен написать что-то большее, путается в коде, а другие, в том числе и профессионалы, его не понимают в силу спутанности его мышления и неумения внятно выразить мысли на уровне современных стандартов. Чтобы преодолеть этот барьер, приходится серьезно и самостоятельно переучиваться – иногда будучи уже студентом или аспирантом. Либо смириться и "носить кофе программистам".

Искусство программирования – это искусство мышления, не надо это забывать, уважаемые школьники, студенты и особенно преподаватели.

Принципы, заложенные в TXLib для повышения качества обучения

- **Сделай сам.** В TXLib многие вещи сделаны или оставлены не совсем удобными для применения. Это – предложение подумать, как сделать это самому, и, как правило, для этого в TXLib есть средства. Сделав, покажите решение другим, если они быстро поймут его и оценят – то ваше решение удачное.
- **Загляни в Help.** (Слово неспроста выбрано английским, потому что большинство информации в современном программировании – на английском языке. Учите его.) Под системой помощи понимается не только TXLib Help, но и весь Internet.
- **Посмотри, как сделано. Загляни в код (см. "Исходные тексты").** Он создавался в том числе как пример программной системы со своей логикой и со своей реализацией, а некоторые функции можно понять только по коду, потому что их нет в системе помощи. Не всегда решения, примененные в TXLib оптимальны даже с точки зрения автора – он надеется, что это убережет желающих научиться качественно, но, увы, нетерпеливых учеников, от ~~Ctrl+C и Ctrl+V~~ плагиата.
- **Посмотри, как сделано *иначе.*** TXLib – не единственная графическая библиотека, и реализация "простого графического холста", примененная в ней – не единственное решение. Посмотри, как устроены десятки других графических библиотек. Но *избегай плохого кода* (его можно определить по тому, как морщатся профессионалы, глядя на него, если у вас нет более объективных средств такого определения) – он научит вас плохому. Хороший, но сложный код (глядя на него, профессионалы не морщатся, а *вздыхают*) – отложи до времени и вернись к нему позже.
- **Выйди за пределы "песочницы".** Это усиление принципа "сделай сам". Собери вместе свои мысли про хорошую библиотеку, посмотри, как устроен TXLib и его аналоги, сделай свою библиотеку, лучше TXLib'а. Это нетрудно. :) Примеры таких библиотек можно найти на сайте TXLib и в Интернете, и некоторые из них сделаны как раз начинающими.

Чтобы научиться программировать – надо программировать. Давайте начнем. Удачи, и May the Source be with you! :)

[1] Погуглите "Литературное программирование" Д. Кнута (D. Knuth, Literate programming).

0. Простейшая программа

Простейшая программа на C (или C++), как и на многих других языках, состоит из двух частей: раздела подключения библиотек и главной функции программы. Рассмотрим пример, в котором рисуется на экране простой рисунок.

0.1. Раздел подключения библиотек

Библиотеки нужны, чтобы использовать команды, определенные в них. Дело в том, что язык C – очень компактный, минималистичный, это облегчает его изучение и использование на разных системах, но без библиотек он содержит мало возможностей. Поэтому для многих действий, например, для работы с графикой, нужны соответствующие библиотеки. Библиотека TXLib позволяет очень легко работать с графикой в Windows, облегчая изучение программирования.

Директивы (команды) подключения библиотек находятся обычно в начале файла программы и выглядят обычно так:

```
#include <stdlib.h>

#include "TXLib.h"
```

TXLib.h и stdlib.h – файлы библиотек, которые подключаются к вашей программе. После этого вы можете использовать команды, функции и переменные, объявленные в этих файлах. Больше, чем нужно, библиотек подключать не стоит, хотя это и не вредно. Когда используется много библиотек, этот раздел может быть большим.

0.2. Главная функция программы

Программа на C (C++) состоит из функций. Функция - это описание каких-либо действий с заданным именем (названием).

Например,

```
int main()
     {
     txCreateWindow (800, 600);

     txLine (320, 290, 320, 220);

     return 0;
     }
```

Главная функция - это функция, с которой начинается исполнение программы. Ее имя - main(). Именно маленькими (строчными) буквами, в Си они отличаются от больших (прописных). Круглые скобки показывают, что речь идет именно о функции, т.е., об описании каких-то действий. Для того, чтобы функция начала работу, ее нужно вызвать. Функцию main()

вызывает сама операционная система компьютера. Слово int означает, что main() в конце работы обязана передать тому, кто ее вызывал (операционной системе), некое целое число. Это число для функции main() означает код завершения нашей программы. Если он равен 0, то работа программы считается успешной.

Действия, записанные в функции, заключаются в фигурные скобки: { и }. Они обозначают начало и конец функции.

Внутри функции записаны вызовы команд, которые рисуют фигуры на экране. Между командами, там, где это логически необходимо, стоят пустые строки, отделяющие одни части программы от других. Это способствует большей понятности программы. Пустая строка в программировании соответствует абзацному отступу в тексте в русском языке.

Последний оператор, return 0, завершает выполнение программы и возвращает операционной системе, которая запустила нашу программу, код завершения программы – число ноль, означающее, что в программе не произошло никаких ошибок.

Для понимания программы и того, чтобы в ней не появлялись ошибки, очень важно, чтобы в нужных местах в ней стояли пробелы. Обычно их ставят до открывающих круглых скобок, после запятых, до и после знаков операций. Наличие пробелов делает программу приятной на вид и предотвращает напряжение глаз при работе с компьютером. Работа с плохо оформленным текстом программы может нанести вред глазам, снизить зрение.

Пример **плохо** написанной программы (так писать **НЕ** надо):

```
#include "TXLib.h"
 int   main(){
 txCreateWindow(800,600);
 txLine(   320,290,320,220);
  txLine (320, 290,280,350);
   txLine(320, 290,360,350)  ;
  txLine(320,230,270,275);
 txLine(320,230, 400,220) ;
 txCircle( 320,190,30)  ;
   txSelectFont( "Times",60);
 txTextOut(240,400,"Hello, world!");
           return        0;
 }
```

```
FFFFF   FFFFF   FFFFF   UU  UU    UU  UU    UU  UU    UU  UU    !!   !!   !!
FF      FF      FF      UU  UU    UU  UU    UU  UU    UU  UU    !!   !!   !!
FF      FF      FF      UU  UU    UU  UU    UU  UU    UU  UU    !!   !!   !!
FFFFF   FFFFF   FFFFF   UU  UU    UU  UU    UU  UU    UU  UU    !!   !!   !!
FF      FF      FF      UU  UU    UU  UU    UU  UU    UU  UU    !!   !!   !!
FF      FF      FF      UU  UU    UU  UU    UU  UU    UU  UU    !!   !!   !!
FF      FF      FF      UU  UU    UU  UU    UU  UU    UU  UU
FF      FF      FF       UUUU      UUUU      UUUU      UUUU     !!   !!   !!
```

Очевидно, что таким стилем программирования зрение будет быстро и безвозвратно испорчено, а мозг уничтожен. :(

Кроме того, ваши будущие коллеги будут вас очень “уважать” за неряшливый и бесструктурный, переусложненный код. “Уважать” – это ирония, на самом деле они даже не захотят с вами работать. А время одиночных проектов в программировании безвозвратно прошло.

Для задания положения рисуемых фигур используется координатная система, у которой начало координат расположено слева-вверху, а ось OY смотрит вниз. Это несколько непривычно, но так традиционно принято в компьютерной графике, поэтому деваться некуда. :) Например, команда

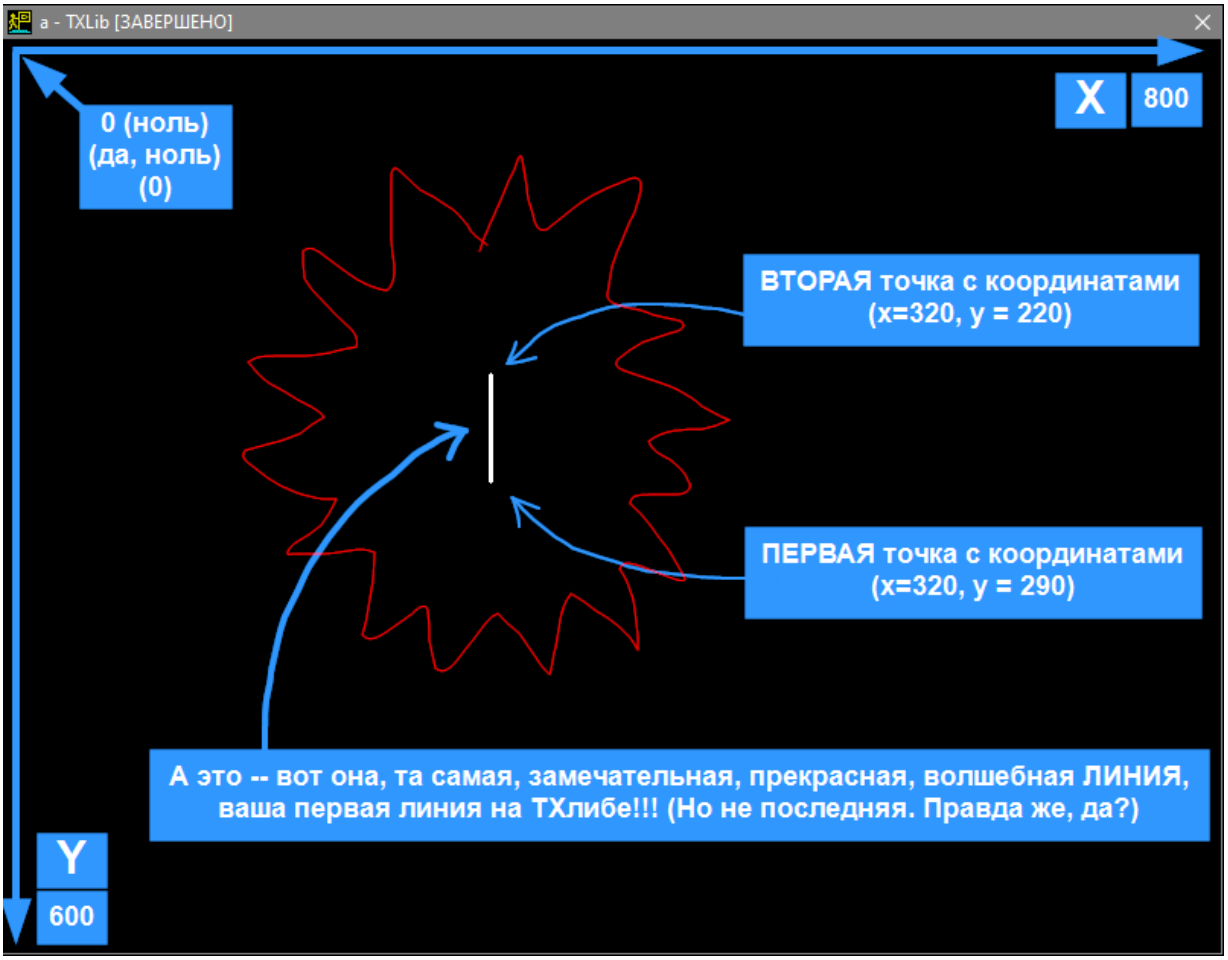
```
txLine (320, 290, 320, 220);
```

Вот из этой программы:

```
#include "TXLib.h"

int main()
    {
    txCreateWindow (800, 600);

    txLine (320, 290, 320, 220);
    }
```

проводит линию из точки x=320 и y=290 в точку с x=320 и y=220.

Та-дааам!.. Вот как выглядит этот по-настоящему потрясающий результат:



Каждая команда заканчивается точкой с запятой. Это – аналог точки в русском языке. Точка с запятой ставится в конце каждого законченного действия или объявления. Например, в строке с txCreateWindow (800, 600) точка с запятой ставится, т.к. в этой строке "закончена мысль" о том, что надо создать окно для рисования. В строке с int main() - не ставится, т.к. описание функции main() не закончено (на самом деле, оно там только начато).

Если в программе используются строки, они заключаются в двойные кавычки, например:

```
    txTextOut (240, 400, "Hello, world!");
```

Если в программе присутствуют числа с дробной частью, то эта часть отделяется от целой части точкой, а не запятой, как в русском языке.

0.3. Что на самом деле происходит при работе с программой

Когда вы пишете код программы, вы его сохраняете в виде текста в текстовом файле, в виде последовательности строк и символов. Тип (так называемое расширение) этих файлов – .CPP, но по сути это текст, который можно открыть даже в обычном "Блокноте". Теперь вспомните школьный курс информатики: за исполнение программ в компьютере отвечает центральный процессор. Он не понимает команд в текстовом виде, так как распознавать текстовые команды – очень медленный и неэффективный процесс, и поэтому эта возможность в него не заложена. Вместо этого команды процессора закодированы номерами, числами, в специальном виде – как говорят, в двоичном формате, и работа с ними происходит гораздо быстрее. Но человеку трудно писать сразу в двоичном формате, поэтому существуют программы-переводчики, транслирующие текстовые файлы с программами в двоичный код машинных команд процессора. Эти программы называются компиляторами, процесс перевода – компиляцией, а события, при этом происходящие – событиями времени компиляции. Для языков Си и C++ самые распространенные компиляторы называются g++ и clang. Если при компиляции происходит что-то не так, то возникает ошибка времени компиляции, машинный код не создается, и его, соответственно, нельзя запустить.

Если ошибок нет, машинный код сохраняется на диске в файле с типом "приложение" (с расширением .EXE) и его уже можно запустить. За запуск программы отвечает операционная система компьютера. По команде пользователя она загружает программу в оперативную память и запускает ее, чтобы процессор мог выполнить ее машинный код команда за командой. То, что происходит при этом, называется событиями времени исполнения. Часто при этом происходят ошибки времени исполнения, которые найти труднее, чем ошибки времени компиляции. Эволюция современных языков программирования идет в направлении большего контроля и проверок кода при компиляции, чтобы не доводить до ошибок времени исполнения. Не стоит этим пренебрегать, в частности, если вместо ошибки при компиляции вы получаете предупреждение (формально не считающееся ошибкой), то лучше не запускать получившуюся программу, а понять и устранить предупреждения.

Когда выполняется оператор "return 0", расположенный в main, выполнение программы завершается, операционной системе возвращается числовой код завершения программы, равный нулю, что означает, что программа завершилась нормально, без ошибок.

0.4. Пример очень маленькой программы

Текст программы

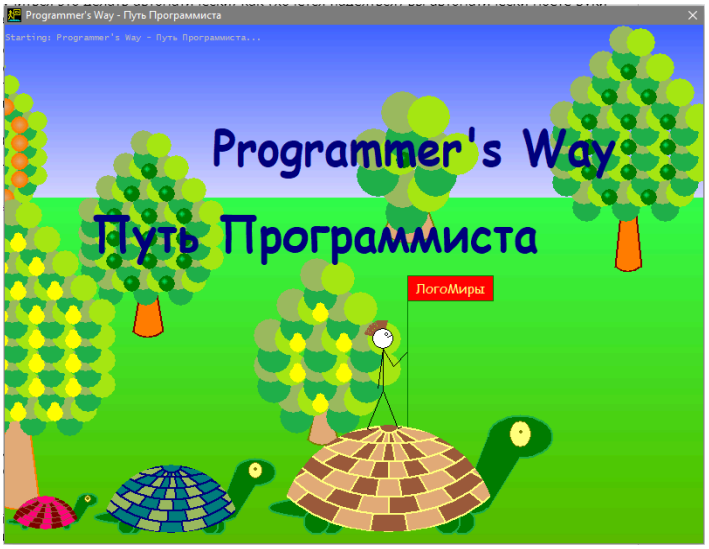
```
#include "TXLib.h"

int main()
    {
    txCreateWindow (800, 600);


    txLine   (320, 290, 320, 220);
    txLine   (320, 290, 280, 350);
    txLine   (320, 290, 360, 350);
    txLine   (320, 230, 270, 275);
    txLine   (320, 230, 400, 220);

    txCircle (320, 190, 30);

    txSelectFont ("Times New Roman", 60);
    txTextOut (240, 400, "Hello, world!");

    return 0;
    }
```

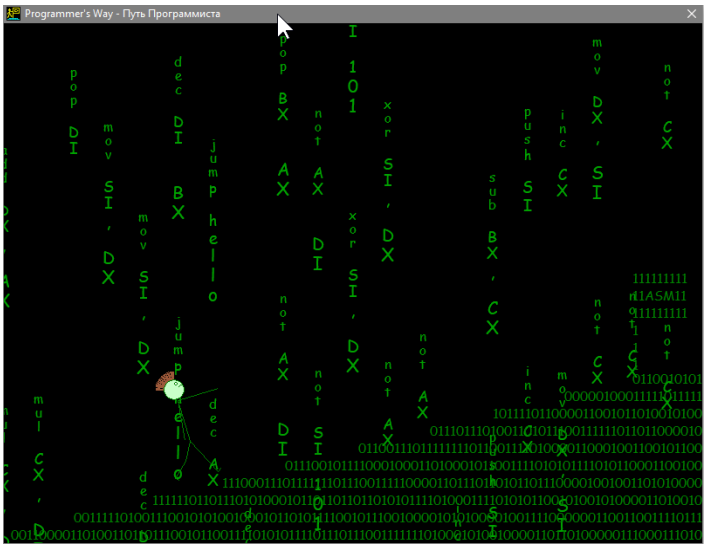
Примечания

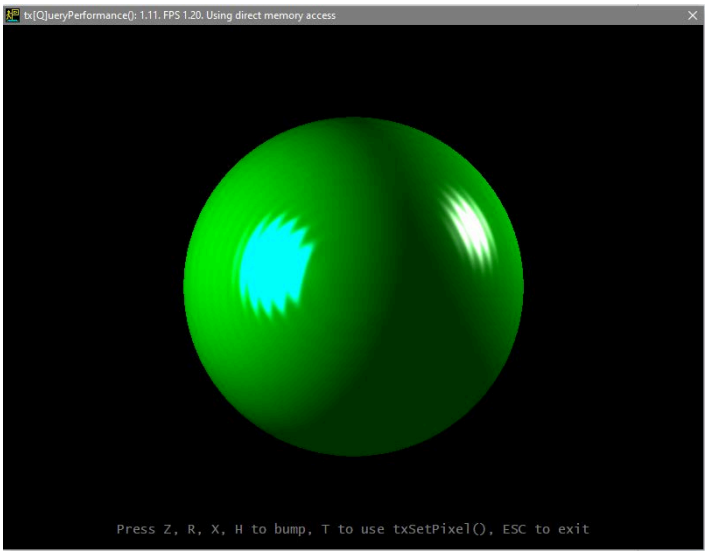
```
Подключение библиотеки рисования

Заголовок главной функции
Начало функции
Создание окна размером 800 на 600 пикселей


Проводится линия из точки x=320 и y=290 в точку с x=320 и y=220
Проводятся другие линии,
    …в результате чего
    …на экране
    …появляется рисунок

Рисуется круг с центром x=320 y=190 и радиусом 30

Выбирается шрифт "Times" размера 60
Печатается текст "Hello, world!" в точке x=240 y=400

Возврат успешного кода завершения программы (0 = нет ошибок)
Конец функции
```

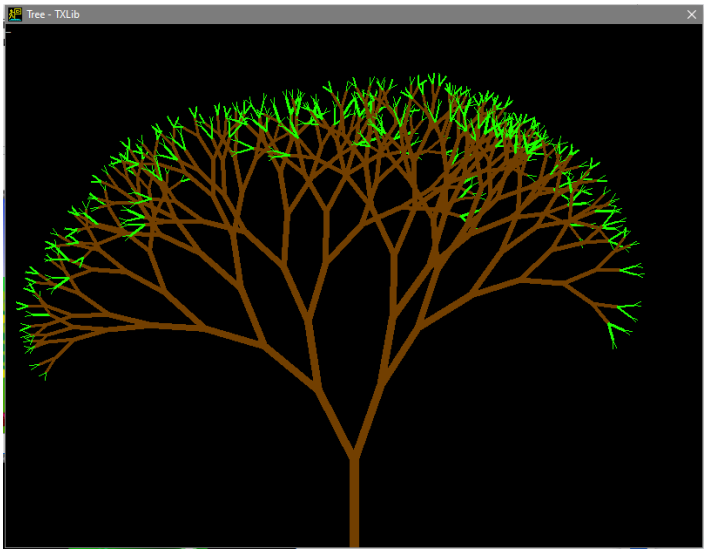
Если вам непонятно, что означает та или иная команда в программе, посмотрите в системе помощи TXLib Help Простейший пример или помощь по этой команде в разделе Рисование. Например, по функциям txLine или txTextOut.

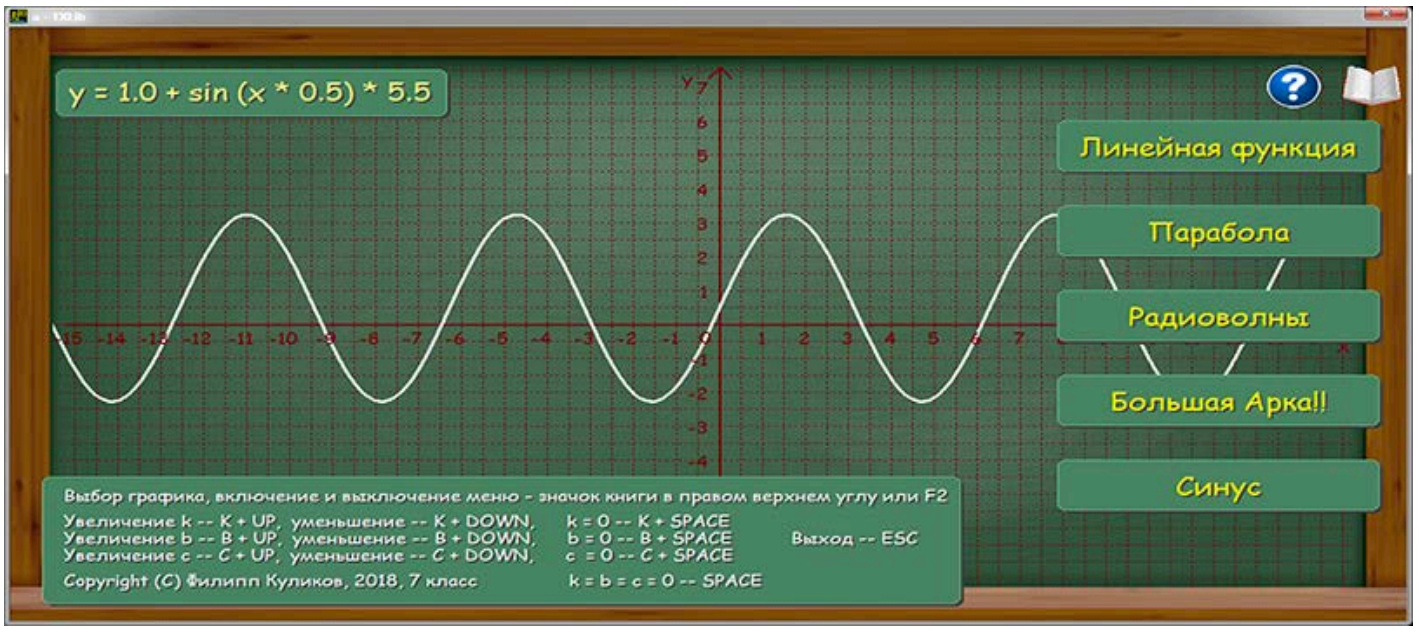
TXLib способен на многое, но только под вашим управлением. Программы сами себя не напишут. Ниже скриншоты программ из примеров к TX Library, написанных в основном на 1-2 году обучения учениками 7-9 классов.



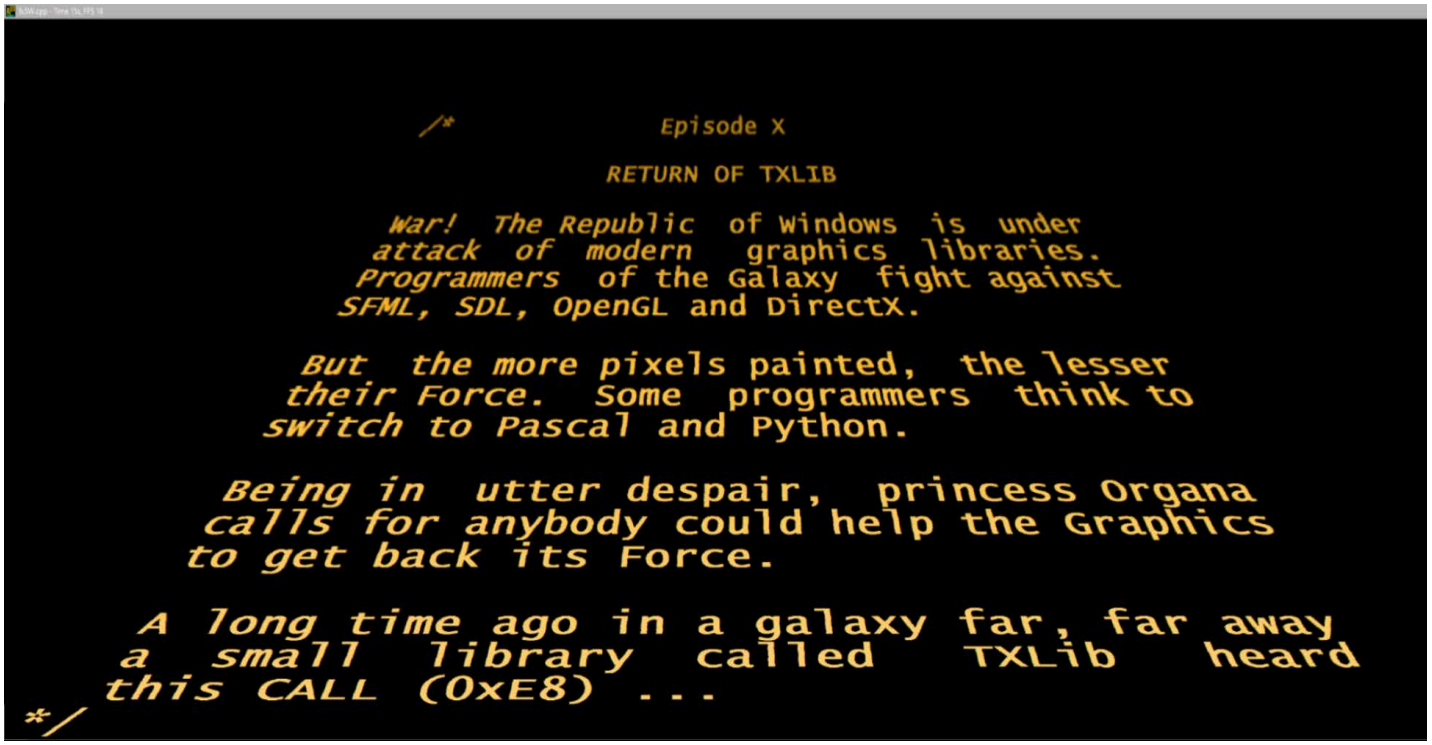








Ну, и если вы фанат Звездных войн:

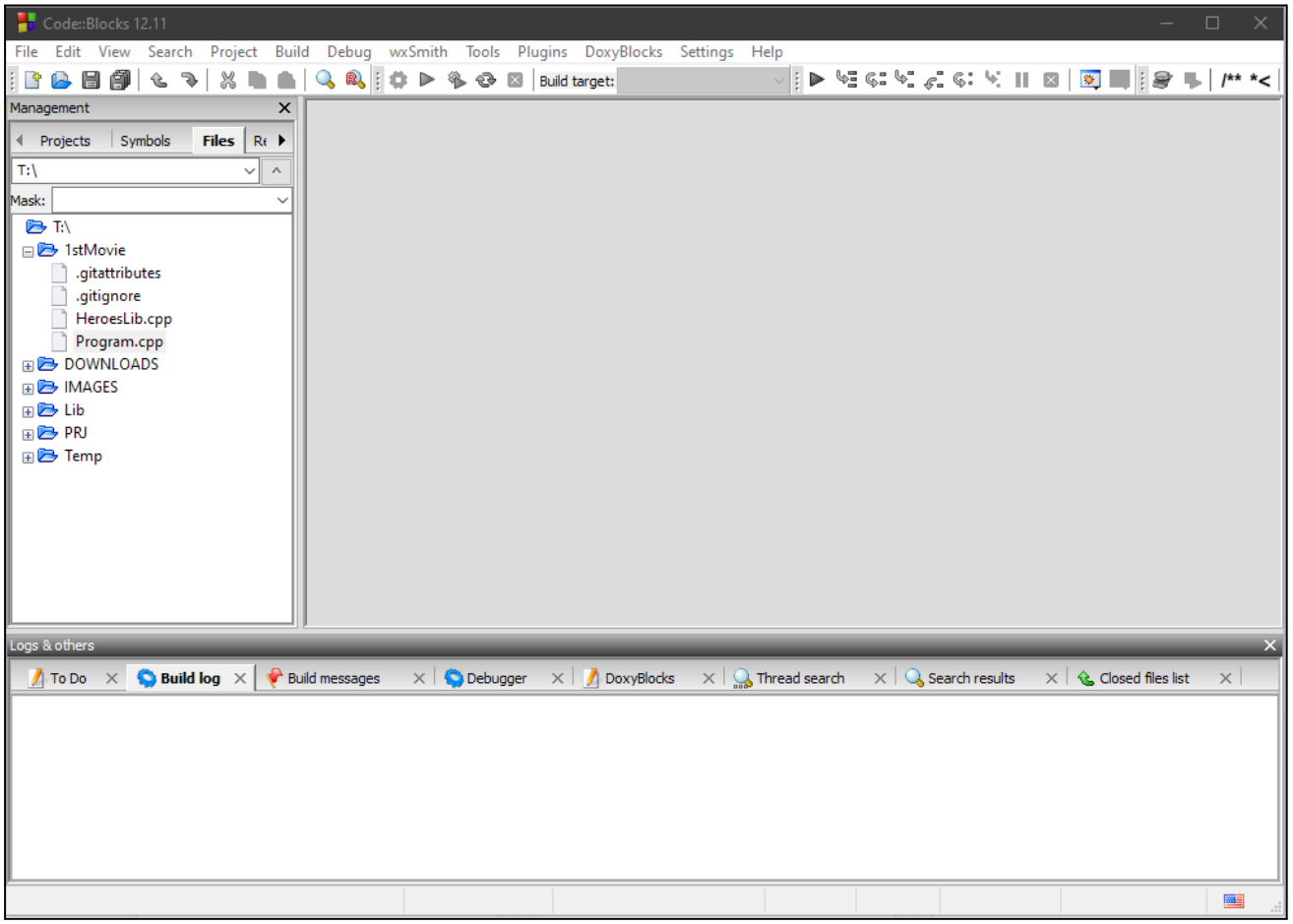


0.5. Где писать код?

Программы для работы с кодом называются "среды программирования" (интегрированные среды разработки, IDE) и представляют собой текстовые редакторы с возможностью удобно компилировать и запускать программы. Иногда такой средой программирования бывает обычный "Блокнот", но это неудобно. Многие пользуются набирающей популярность средой программирования VS Code, но она сложна в настройке для начинающих. Для начинающих удобна сборка среды программирования CodeBlocks, которую можно скачать по адресу storage.ded32.net.ru/Lib/TX/codeblocks-mingw-setup-rar.exe или gg.gg/TXLib-CodeBlocks. В нее добавлена урезанная версия TXLib (без примеров и документации, так что лучше скачайте полную версию с официального сайта txlib.sf.net) и много чего еще полезного, а вирусов там нет, как бы ни предупреждала недоверчивая Microsoft. Настройки стандартной сборки с официального сайта CodeBlocks хуже, использовать их не стоит. Там неудобный шрифт, с которым трудно находить опечатки, много лишних панелей, неоптимально занимающих место на экране, другие проблемы.

Лучше устанавливать сначала CodeBlocks, затем TXLib. Устанавливать лучше в папки, в которые есть доступ по чтению и записи (не C:\Program Files...). Иначе в будущем может потребоваться доступ в режиме администратора, например, если вам захочется обновить компилятор.

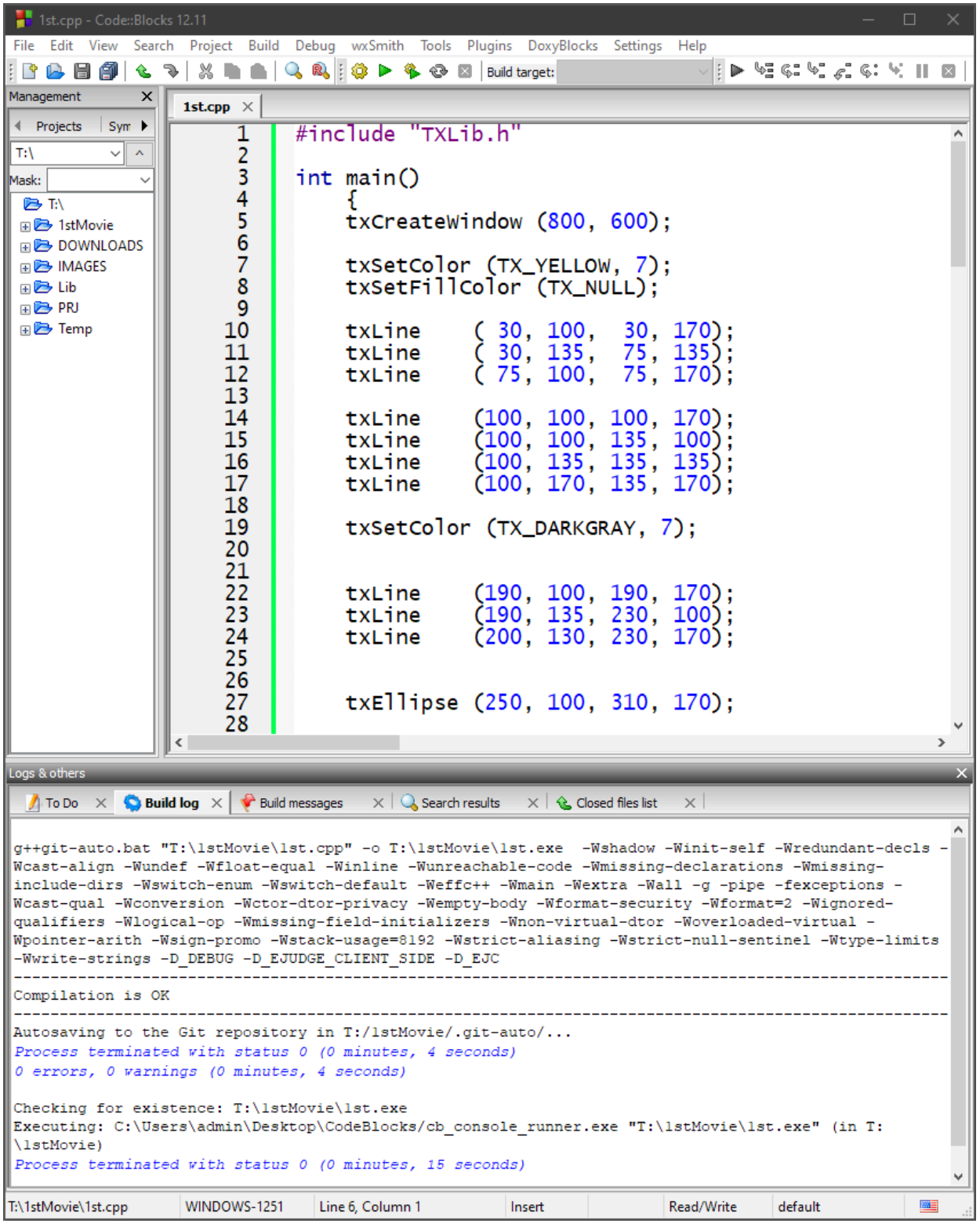
После установки CodeBlocks или запуска с помощью ярлыка на рабочем столе видим такую картину:



Нажимаем Ctrl+Shift+N, создается новый файл, там пишем код, сохраняем, как обычно, через Ctrl+S. Не надо мышкой искать всякие кнопочки, их нажимать долго, учите горячие клавиши, вы же программисты. Имя для файла выбирайте не какое-то там Untitled, то есть безымянный. Фу. Назовите нормально, например, This_is_my_first_program_written_in_CPP_using_TXLib_I_love_programming_a_lot_but _less_than_my_sandwich.cpp, придумайте что-то свое, покороче, но понятное. Не надо включать в имя пробелы, скобки и другие специальные символы (символ подчеркивания – можно). Иначе могут быть проблемы с программой, если не сейчас, то потом. Несохраненный файл помечается звездочкой рядом с его именем в заголовке вкладки. Бойтесь этой звездочки. Если все повиснет-перезапустится-обновится, как это водится в Windows, то файл сам собой не сохранится. Все придется набирать заново, с момента последнего сохранения. Также подумайте, куда будете сохранять свое творение, чтобы не искать потом по всему диску (а оно на флешке, забытой дома).

Набрав (не забудьте Ctrl+S) в верхнем окне (Ctrl+S) текст программы (Ctrl+S), нажимаем Ctrl+F9 – это компиляция. По этой команде запускается компилятор языка C++ – это программа, переводящая текст на C++, который вы пишете, в машинный код (исполняемый файл). Запустить исполняемый файл можно клавишей Ctrl+F10. Объединяет эти две команды клавиша F9, но так вы можете не увидеть предупреждения компилятора о возможных ошибках, и потом будете искать их долгими ночами во время выполнения программы.

Вот как выглядит окно CodeBlocks после набора текста программы, сохранения его в файл (Ctrl+S), компиляции (Ctrl+F9) и запуска (Ctrl+F10):



В нижней части окна вкладки Build Log и Build Messages показывают ошибки и предупреждения компиляции. При ошибках исполняемый код не создается и новую версию программы не запустить. Предупреждения говорят о скрытых ошибках, поэтому их надо обязательно читать и исправлять. Если ошибок и предупреждений нет, то можно наслаждаться непревзойденной графикой TXLib под вашим чутким творческим руководством.

1. Первый проект: мультфильм

1.1. Начальное задание

В первом проекте мы будем делать, то есть писать код, простейшего компьютерного мультфильма. Сюжет может быть любой, но одно условие – он должен быть вам интересен. Для начала придумайте героев вашего мультфильма. "Героем" может быть любой объект, состоящий из шести или более геометрических фигур разных цветов. Меньше шести фигур не надо, больше – лучше. Например, дерево, состоящее из пяти линий и двух окружностей подойдет. Зачем дереву окружности? Это могут быть яблоки на яблоне или, скажем, сосна с глазами (вот так вот, да). Но отлично подойдет и для кроны дерева. А недорисованный человек, состоящий из четырех линий и одной окружности - не герой (не объект). Всего героев (объектов) должно быть шесть или более. Меньше – не надо, больше – лучше. Увлеченные люди делают много героев из многих фигур, им трудно остановиться. Но, правда, спать когда-то надо. Герои (объекты) должны быть принципиально не похожи друг на друга. Не надо рисовать похожих людей или предметов, в разных позах, разных размеров или разных цветов и считать их разными героями. "Размножить" героя, изменить его позу и цвет мы легко сможем в последующих главах. Сейчас такое "размножение вручную" будет бесполезным, не тратьте на него силы. Лучше придумайте еще разных новых героев.

Далее нарисуйте всех этих героев на одном общем рисунке на горизонтальном листе клетчатой бумаги формата А4, или на развороте тетради, или, лучше, в редакторе Microsoft Paint. Измерьте расстояние до каждой точки от левого и верхнего краев картинки (в MS Paint в левом нижнем углу окна оказываются координаты мышиного курсора, это очень удобно). Рисунок начнет походить на чертеж. Не рисуйте героев на разных рисунках, нам они нужны все вместе.

Далее напишите **одну** программу, которая рисует всех героев в одном окне, используя команды (функции) библиотеки TXLib: txLine, txCircle, txSetColor, txSetFillColor и многие другие из раздела рисование системы помощи библиотеки TXLib. Внимание: не надо пользоваться переменными, циклами, комментариями, функциями, массивами и другими конструкциями, если вы знаете, что это такое. Сейчас для работы это вредно. Если не знаете – и прекрасно, не надо сейчас это выяснять, вы это найдете в книге чуть позже, проявите немного терпения. Пишите просто команды TXLib, это называется линейным алгоритмом. Строго соблюдайте стиль кода: пишите аккуратно, ставьте побольше пробелов для читаемости текста (но не бессистемно, а логично, единообразно). Сдвиньте весь текст, принадлежащий main, на четыре пробела вправо, для этого достаточно в начале строки нажать клавишу Tab (или выделить фрагмент текста и нажать Tab или Shift+Tab, попробуйте). Там, где проходят границы между частями программы, обязательно ставьте пустые строки. Иначе запутаетесь. Пусть структура программы ярко и четко бросается вам в глаза, и ничто не мешает читать ее текст. У вас получится примерно такое:

```
#include "TXLib.h"
```

```
int main()
    {
    txCreateWindow (800, 600);

    txSetColor (TX_YELLOW, 7);
    txSetFillColor (TX_NULL);

    txLine    ( 30, 100,  30, 170);
    txLine    ( 30, 135,  75, 135);
    txLine    ( 75, 100,  75, 170);

    txLine    (100, 100, 100, 170);
    txLine    (100, 100, 135, 100);
    txLine    (100, 135, 135, 135);
    txLine    (100, 170, 135, 170);

    txSetColor (TX_DARKGRAY, 7);

    txLine    (190, 100, 190, 170);
    txLine    (190, 135, 230, 100);
    txLine    (200, 130, 230, 170);

    txEllipse (250, 100, 310, 170);

    txSetColor (TX_GRAY, 7);

    txLine    (335, 100, 335, 170);
    txLine    (335, 100, 380, 100);
    txLine    (380, 100, 380, 170);

    txLine    (410, 100, 410, 170);
    txLine    (410, 170, 460, 100);
    txLine    (460, 100, 460, 170);

    txSetColor (TX_LIGHTGRAY, 7);

    txLine    (490, 100, 490, 170);
    txArc     (450, 100, 530, 140, 270, 180);

    txLine    (540, 100, 570, 145);
    txLine    (590, 100, 560, 170);

    txSetColor (TX_WHITE, 7);

    txLine    (610, 100, 610, 170);
    txLine    (610, 170, 660, 100);
    txLine    (660, 100, 660, 170);
    txLine    (635,  90, 620,  80);
    txLine    (635,  90, 650,  80);

    txLine    (690, 100, 690, 150);
    txCircle  (690, 170, 2);

    txLine    (710, 100, 710, 150);
    txCircle  (710, 170, 2);

    txLine    (730, 100, 730, 150);
    txCircle  (730, 170, 2);

    txSetColor (TX_LIGHTGREEN, 7);

    txLine    ( 50, 230,  50, 300);
    txLine    ( 50, 230, 100, 230);
    txLine    (100, 230, 100, 300);
```

```
    txLine    (130, 230, 130, 300);
    txLine    (130, 300, 180, 230);
    txLine    (180, 230, 180, 300);

    txSetColor (TX_LIGHTCYAN, 7);

    txLine    (210, 230, 210, 300);
    txLine    (240, 230, 240, 300);
    txLine    (270, 230, 270, 300);
    txLine    (210, 300, 270, 300);

    txLine    (300, 230, 300, 300);
    txLine    (300, 300, 350, 230);
    txLine    (350, 230, 350, 300);

    txSetColor (RGB (255, 128, 128), 7);

    txArc     (400, 230, 460, 300, 45, 270);

    txLine    (500, 230, 470, 300);
    txLine    (500, 230, 530, 300);
    txLine    (485, 275, 515, 275);

    txLine    (550, 300, 550, 230);
    txLine    (550, 230, 580, 300);
    txLine    (580, 300, 610, 230);
    txLine    (610, 230, 610, 300);

    txCircle  (650, 250, 3);
    txCircle  (650, 295, 3);

    txArc     (330, 140, 690, 400, 345, 30);
    }
```

Обратите внимание, что в примере нет ни одного объекта, который может считаться полноценным героем. Почему? Если считать, что объекты ограничиваются пустыми строками (рисование одного объекта – один абзац текста), то в каждом из них не более четырех-пяти фигур. А надо шесть фигур или более.

Вот в этом примере уже есть два героя, по 10 и по 6 фигур:

```
#include "TXLib.h"

int main()
    {
    txCreateWindow (800, 600);

    txSetColor (TX_WHITE);
    txSetFillColor (TX_TRANSPARENT);
    txRectangle (10, 10, 790, 590);

    txSetColor (TX_LIGHTCYAN);
    txEllipse (200, 150, 600, 450);
    txEllipse (245, 150, 555, 450);
    txEllipse (290, 150, 510, 450);
    txEllipse (330, 150, 470, 450);
    txEllipse (375, 150, 425, 450);
    txEllipse (200, 150, 600, 450);
    txEllipse (200, 190, 600, 410);
    txEllipse (200, 230, 600, 370);
    txEllipse (200, 270, 600, 330);
    txLine    (200, 300, 600, 300);
```

```
    txSetColor (TX_LIGHTGREEN);
    txSelectFont ("Times New Roman", 60);
    txSetTextAlign (TA_CENTER);
    txTextOut (400, 480, "Hello, world!");

    txSetColor (TX_YELLOW);
    txSetFillColor (TX_YELLOW);
    txLine   (385, 135, 385, 120);
    txLine   (385, 135, 375, 150);
    txLine   (385, 135, 395, 150);
    txLine   (385, 125, 375, 135);
    txLine   (385, 125, 400, 120);
    txCircle (385, 115, 6);

    txSetFillColor (TX_TRANSPARENT);
    txLine (400, 75, 400, 150);
    txRectangle (400, 75, 450, 115);
    txSelectFont ("Times New Roman", 20);
    txTextOut (425, 85, "C++");

    txTextCursor (false);
    return 0;
    }
```

Копировать этот текст в редактор кода и выдавать за свою домашку не надо, вам станет неловко. :) Сделайте задание сами. Не ленитесь.

...

Сделали задание?

После него подумайте вот над чем. Какие действия в процессе создания программы вам показались наиболее неприятными и утомительными? На первый взгляд вопрос неожиданный, но важный. В труде программиста всегда бывают вещи, не очень удобные технологически, их надо уметь замечать, выяснять их причины и устранять. Лучше обсудить это с друзьями, с которыми вы вместе учитесь. Ответы на этот вопрос понадобится нам очень скоро, в главе “Функции”.

Теперь поговорим о разного рода ошибках, потому что поначалу у вас их будет много.

1.1.1. Распространенные ошибки и борьба с ними

Первое правило в борьбе с ошибками: не бойтесь их количества. Часто первые ошибки влекут за собой последующие. И если исправить одну, то последующие могут пропасть, т.к. являются её следствием.

Поэтому *второе правило:* прежде всего анализируйте и исправляйте *самую первую* ошибку.

Третье правило: будьте внимательны. Об этом см. ниже.

Четвертое правило ~~волшебника:~~ ищите ошибку вначале у себя, а только потом в библиотеках. Библиотеки достаточно долго тестировались, поэтому вероятность ошибки в них очень мала.

Ошибки, не связанные с текстом программы

- Вначале убедитесь, что файл с программой имеет правильное расширение ".CPP", а не расширение ".C". Если расширение (тип) файла неправильное, то в списке ошибок одной из первых будет сообщение *"TXLib.h: Must use C++ to compile TXLib.h".* Это потому, что для библиотеки требуется компилятор C++, а он транслирует файлы с расширением ".CPP". Файлы ".C" транслирует компилятор Си, а не C++.
  Чтобы сменить расширение файла, выберите в главном меню [File], [Save As] и дайте программе имя с расширением ".CPP". Лучше не использовать ни русских букв, ни пробелов в имени файла.

- Если файл библиотека TXLib установилась неверно, то файл TXLib.h может быть не найден. Сообщение об ошибке зависит от компилятора, но будет содержать фразу, подобную *"TXLib.h file not found".* В этом случае выйдите из среды программирования и переустановите TXLib. Если это не помогает, то просто скопируйте файл TXLib.h из папки TX в папку с примерами или в папку с вашей программой.

Ошибки, связанные с текстом программы

В простых программах ошибки бывают двух видов:

- Ошибки записи (орфография, пунктуация) - их называют *синтаксическими ошибками* (syntax error). Они происходят до запуска программы, на стадии перевода программы в машинный код (стадии компиляции). Поэтому их называют *ошибками времени компиляции* (compile-time errors). Исполняемый файл при этом **не** создается и программа **не** запускается.

- Логические ошибки - они происходят после запуска программы, при этом при компиляции ошибок нет (иногда имеются предупреждения, warnings, которые полезно исправлять, а лучше не допускать их появления). Их называют *ошибками времени исполнения* (runtime errors).

Ошибки времени компиляции

Практически все синтаксические ошибки на этой стадии происходят из-за невнимательности. Распространенные синтаксические ошибки:

Путают ключевые слова, названия библиотек и команд:

```
#iclude "TX lib.h"              Правильно: #include "TXLib.h"
in maim()                       Правильно: int main()
tx Line (10, 10, 20, 20)        Правильно: txLine (10, 10, 20, 20);
```

Путают большие и маленькие буквы:

```
txcircle (100, 100, 10)         Правильно: txCircle
```

Не ставят скобки:

```
int main                        Правильно: int main()
```

Забывают запятые или путают их с другими знаками:

```
txCircle (100  100  10)          Правильно: txCircle (100, 100, 10);
txCircle (200; 200; 20)          Правильно: txCircle (200, 200, 20);
```

Забывают точки с запятой:

```
txSelectFont ("Times", 60)       Правильно: txSelectFont ("Times", 60);
```

Забывают или путают кавычки:

```
txSelectFont (Times, 60)         Правильно: txSelectFont ("Times", 60);
txSelectFont ('Arial', 20)       Правильно: txSelectFont ("Arial", 20);
```

Ставят лишние точки с запятой там, где "мысль не закончена":

```
int main();                      Правильно: int main()
```

Указывают дробную часть числа не через точку, а через запятую:

```
3,1415                           Правильно: 3.1415
```

Забывают фигурные скобки:

```
Правильно:
                                 int main()
int main()                           {
txCreateWindow (800, 600);           txCreateWindow (800, 600);
txLine (320, 290, 320, 220);         txLine (320, 290, 320, 220);
return 0;                            return 0;
                                     }
```

Забывают писать открывающие или закрывающие скобки, или пишут лишние, отчего появляются непарные скобки, или путают виды скобок:

```
int main()
    {
    txCreateWindow (800, 600);

    txLine (320, 290, 320, 220;  Ошибка: Не закрыта круглая скобка
    return 0;
    }

int main()
{
txCreateWindow {800, 600};       Ошибка: Фигурные скобки вместо круглых
return 0;
}

txLine (320, 290, 320, 220);     Ошибка: Команда за пределами функции main()
}                                Ошибка: Лишняя скобка
```

Помещают команды за пределы фигурных скобок

```
int main()
{
txCreateWindow (800, 600);
return 0;
}
```

```
    txLine (320, 290, 320, 220);      Ошибка: Команда за пределами функции main()
```

Указывают лишние данные в командах или указывают меньше данных (аргументов), чем нужно.

При таком несоответствии количества параметров (too few arguments... - слишком мало аргументов, или too many arguments... - слишком много) среда программирования часто указывает на правильное определение той команды, которая была неверно вызвана. Это нужно для того, чтобы вы посмотрели на это определение и вспомнили, сколько данных надо передавать. Это совсем не означает "ошибки в библиотеке" или "ошибки в стандартной команде". Настоящее место ошибки там, где команда вызвана. В это место легко попасть, если кликнуть мышкой 2 раза на строку с надписью "...at this point in file" в списке ошибок, в нижней части окна среды программирования.

Примеры таких ошибок:

```
    txCircle (100, 100);            Здесь не указан радиус
    txCircle (200, 200, 20, 30);    Здесь лишнее число - txCircle() принимает всего 3 аргумента
```

Ошибки времени исполнения

Бывают и логические ошибки, которые не всегда находит компилятор. Например, вы не выбрали нужный цвет рисования (по умолчанию он белый), или цвет совпал с фоном и поэтому не виден. Или вы нарисовали одну фигуру поверх другой и она закрыла предыдущую, или задали неверные координаты. Эти ошибки появляются после запуска программы, поэтому их называют *ошибки времени выполнения* (runtime errors). Их легко понять, если выполнить на листе бумаги все команды одну за другой, в том порядке, в котором они заданы в программе. При этом не надо стараться выполнять "так, как лучше", или "как хочется, чтобы получилось". Выполняйте так, как будто это не ваша, а совсем чужая работа, и вам не интересен ее результат. В тот момент, когда результат разойдется с вашим желанием и вы получите странную фигуру, станет понятно место ошибки.

Конечно лучше, если ошибка произошла на стадии компиляции, чем на стадии выполнения, когда искать ее труднее. Поскольку человеку избежать ошибок невозможно, один из видов мастерства программирования состоит в том, чтобы постараться перевести хотя бы часть ошибок из стадии исполнения на стадию компиляции. Пусть они происходят там, где их легче ловить. :)

Ошибки и стиль программирования

Правильное оформление кода (стиль программирования) помогает перевести ошибки стадии выполнения на стадию набора текста, и находить их даже до компиляции. Например, неверные координаты очень сложно проследить в "примере плохо написанной программы" (см. выше). Сделать это гораздо легче в "законченном примере" (см. ниже), где все числа, операторы и команды аккуратно выровнены. Если программа длиннее нескольких строк, то между ее логическими частями ставят пустые строки (вместо "красных строк" в русском языке), это позволяет не напрягать логическое мышление зазря, а концентрироваться на творчестве и на развитии программы.

Оформлять код "красиво" уже после того, как программа написана и отлажена, достаточно бессмысленно – хороший стиль облегчает написание и отладку, а ведь программа уже готова.

:) Поэтому ставить пробелы, пустые строки и выравнивать текст нужно сразу, при наборе, и приучиться это делать автоматически, как (хочется надеяться) вы автоматически моете руки перед едой. :)

Многие опытные люди, желая помочь вам, бывают разочарованы плохим стилем и перестают помогать и советовать. Поэтому чтобы регулярно получать помощь знатоков, нужно держать стиль на высоте. К счастью, оформить программу - это самое легкое дело в программировании, но и одновременно это первый шаг к надежности и мастерству.

1.1.2. Пример, который не надо копировать

Проанализируйте его, но не берите из него героев или заготовок. Работайте полностью самостоятельно.

Почему?

Ну, во-первых, просто копируя, или даже копируя и слегка что-то изменяя, но в целом оставляя, как было, по-настоящему ничему не научишься. Давайте договоримся, что в ваших проектах вы не будете использовать этот пример и похожие на него. Пишите сами.

Во-вторых – формальный момент – здесь слишком мало героев. Но главное – в абзаце выше.

Зайти в папку, куда установлен TXLib (вы ее помните? Если нет, то...), открыть в папке Examples нужный пример (поищите...), запустить его, чтобы посмотреть, как он работает – это ОК, это нормально. Поэкспериментировать с ним, что-то убрать, что-то добавить – да. Но передирать текст или его куски – контрпродуктивно. Вы просто ничему не научитесь, это будет потеря времени. Поработав с примером, закройте его и напишите полностью с нуля сами. Не заглядывая.

Полностью. С нуля. Сами.

Иначе никак.

Этот пример входит в состав примеров TXLib (Пример: Улучшенный).

```
#include "TXLib.h"

int main()
    {
    txCreateWindow (800, 600);

    txSetColor (TX_WHITE);
    txSetFillColor (TX_TRANSPARENT);
    txRectangle (10, 10, 790, 590);

    txSetColor (TX_LIGHTCYAN);
    txEllipse (200, 150, 600, 450);
    txEllipse (245, 150, 555, 450);
    txEllipse (290, 150, 510, 450);
    txEllipse (330, 150, 470, 450);
    txEllipse (375, 150, 425, 450);
    txEllipse (200, 150, 600, 450);
    txEllipse (200, 190, 600, 410);
    txEllipse (200, 230, 600, 370);
```

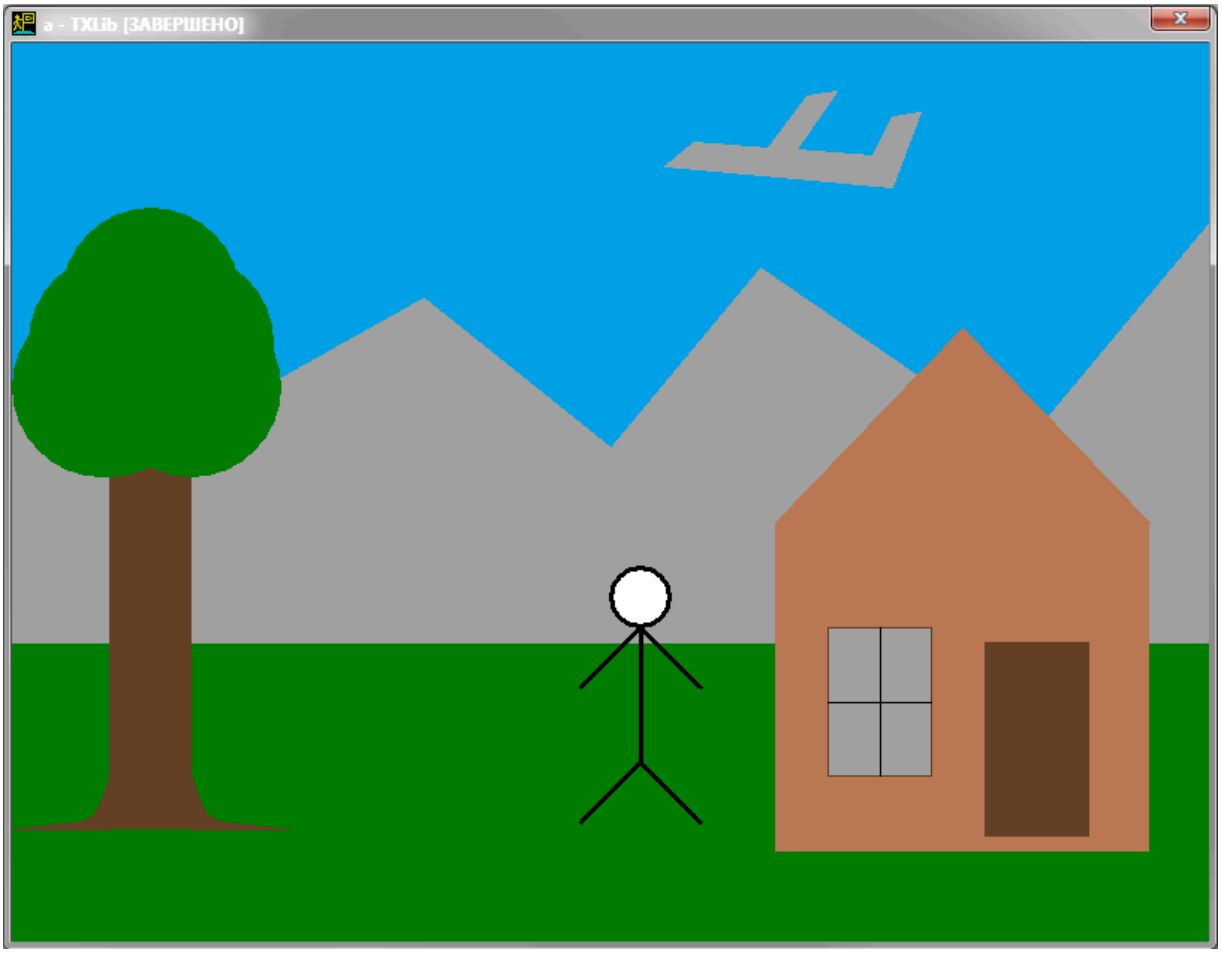
```
    txEllipse (200, 270, 600, 330);
    txLine    (200, 300, 600, 300);

    txSetColor (TX_LIGHTGREEN);
    txSelectFont ("Times New Roman", 60);
    txSetTextAlign (TA_CENTER);
    txTextOut (400, 480, "Hello, world!");

    txSetColor (TX_YELLOW);
    txSetFillColor (TX_YELLOW);
    txLine   (385, 135, 385, 120);
    txLine   (385, 135, 375, 150);
    txLine   (385, 135, 395, 150);
    txLine   (385, 125, 375, 135);
    txLine   (385, 125, 400, 120);
    txCircle (385, 115, 6);

    txSetFillColor (TX_TRANSPARENT);
    txLine (400, 75, 400, 150);
    txRectangle (400, 75, 450, 115);
    txSelectFont ("Times New Roman", 20);
    txTextOut (425, 85, "C++");

    txTextCursor (false);
    return 0;
    }
```

1.1.3. Ревью кода

Ревью кода – это набор советов, как код следует улучшить, указание неоптимальных, неаккуратных, путаных мест, которые надо обязательно устранять. Программирование – одно из немногих занятий, где нам мало что мешает физически: не нужны станки и дорогое производство, не надо покупать материалы, если вы что-то испортили. Многие вещи делаются сильно быстрее и легче, чем в материальном мире. Поэтому давайте не лениться переделывать, и будем доводить свои работы до идеального состояния в каждый момент времени. Помните в предисловии было про перфекционизм и трудолюбие? Вот они тут и нужны. Иначе время пройдет, а толку не будет.

После каждой работы мы обязательно будем делать ревью кода, этому следуют все профессиональные программисты. Во многих компаниях даже недостаточно одного ревью, нужно минимум два. (В отделе программирования NASA, знаменитом тем, что по их вине не случилось ни одной аварии, существуют даже ревью на ревью кода.) В вашем случае, чем больше код прошел ревью у разных людей, тем лучше, потому что способов сделать работу хорошо – много, и вы должны знать их как можно больше. Помните: если вы знаете единственный способ решения задачи, то он неверен. Ищите другие способы, только сравнив их, вы получите право выбрать лучшее. Иногда это "лучшее", кстати, резко меняется при изменении условий задачи. Настоящий инженер всегда придумывает "пучок" решений, и, выбирая одно из них, не забывает о других.

Рассмотрим пример. Пусть есть программа, рисующая такую картинку:



Исходный текст у нее такой:

```
#include "TXLib.h"

int main()
    {

    txCreateWindow (800, 600);

const COLORREF SKY = RGB (0, 162, 232);
txSetFillColor (SKY);
txClear ();

    txSetColor(TX_GREEN);
    txSetFillColor (TX_GREEN);
    txRectangle(0,400, 800,600);

    txSetColor (TX_GRAY);
    txSetFillColor (TX_GRAY);
    POINT mountains[10] = {{-50, 320}, {50, 150}, {150, 240},
{275, 170}, {400, 270}, {500, 150}, {675, 270}, {800, 120}, {800,
400}, {0, 400}};
    txPolygon (mountains, 10);

    const COLORREF HOUSE = RGB (185, 122, 87);
    txSetColor (HOUSE);
    txSetFillColor (HOUSE);
    txRectangle (510, 320, 760, 540);
    POINT a[3] ={{510,320},{635,190}, {760,320}};
```

```
Номер фрагмента

1
1
1
1
1
1
1
1

2
2
2

3
3
3
3
3

4
4
4
4
4
```

```
    txPolygon (a, 3);                                          4
    const COLORREF WOOD = RGB (103, 65, 44);                   4
    txSetColor (WOOD);                                         4
    txSetFillColor (TX_GRAY);
    txRectangle(545,390,615,490);                              5
    txSetFillColor(WOOD);                                      5
    txRectangle(650, 400,720,530);                             5
    txSetColor (TX_BLACK);                                     5
    txLine(545,440, 615,440);                                  5
    txLine(580,390, 580,490);                                  5
    }
```

Что здесь хорошего?

Много что. Код по большей части аккуратный. Пробелы везде стоят так, что текст отлично читается. Пустые строки отделяют фрагменты текста. Из-за того, что функция txPolygon, рисующая многоугольник, требует имени контура (посмотрите систему помощи по ней), немного становится понятен смысл фрагментов текста, за рисование каких объектов они отвечают. Функция txPolygon мощная, и способна нарисовать целый объект за один раз. Однако давайте договоримся: каждое применение этой функции будет считаться за одну фигуру. Поэтому в коде выше у нас всего один полноценный объект, за который отвечает самый последний фрагмент текста. Так как там используется константа для цвета под названием HOUSE (посмотрите, как делать константы в описании к функции RGB и набору цветов txColors), а также название контура attic (чердак), то речь идет о доме. Горы, рисующиеся перед домом, полноценным объектом не являются: они рисуются за один вызов txPolygon, то есть они представляют собой всего лишь одну фигуру. Если бы они рисовались отдельными линиями – конечно, был бы полноценный объект, причем довольно богатый.

Что здесь не совсем так?

Во-первых, в начале фрагмента 1 перед txCreateWindow, возможно, стоит лишняя пустая строка. Создание окна входит в начало программы, акцентировать именно на этой строке, на наш взгляд, избыточно. Если вы с этим согласны, уберите ее, если стало лучше – оставьте как есть. В стиле кода (способе форматирования текста программы) может быть много правил, но они должны применяться логично и всегда для визуального усиления понимания структуры и логики программы. Стилей кода много, ваш собственный только-только зарождается, ориентируйтесь на ваше визуальное удобство и логику. Слепо, формально следовать стилю не надо (пока вы не пошли работать в компанию, где следование общему корпоративному стилю обязательно. Да и там, все равно, логика превыше формальности). Но форматировать абы как, где-то поставив пробел после запятой или перед скобкой, где-то нет, с жаром отстаивая право на индивидуальность, а на самом деле ленясь подумать и исправить, точно не надо. Вы уже поняли, почему: так вы ничему не научитесь.

Если вам нужны примеры хорошего, логичного форматирования, посмотрите на любую сверстанную в типографии и изданную книгу. Попробуйте, например, в любом типографском тексте найти открывающую скобку без пробела перед ней, если она стоит не после другой скобки. Или запятую без пробела после нее, или с пробелом перед ней. Не найдете. Редакторы, корректоры и верстальщики-профессионалы строго следят за соблюдением разных, но глубоко логичных стилей оформления текста, для них это наука, искусство и целая жизнь, а нарушение их – страшный сон и переверстка книги следующим же утром.

Можете посмотреть в Интернете разные стили форматирования кода, но лучше сейчас применять тот, что в книге или близкий к нему. Он называется Whitesmith и идеален для восприятия кода, особенно начинающими.

Во-вторых, и это уже бесспорный дефект, в конце фрагмента 1 отступ текста вправо, обозначающий принадлежность к главной части программы main, исчез. Три строки кода улетели налево и “повисли” в воздухе, как будто они не относятся к главной программе:

```
#include "TXLib.h"

int main()
    {

    txCreateWindow (800, 600);

const COLORREF SKY = RGB (0, 162, 232);
txSetFillColor (SKY);
txClear ();

    ...
    }
```

Иногда в некоторых плохих текстах отступов вправо совсем нет, тогда совсем не видно подчиненности строк друг другу:

```
#include "TXLib.h"

int main()
{

txCreateWindow (800, 600);

const COLORREF SKY = RGB (0, 162, 232);
txSetFillColor (SKY);
txClear ();

...
}
```

Как можно было бы поправить это? Вот так:

```
#include "TXLib.h"

int main()
    {
    txCreateWindow (800, 600);

    const COLORREF SKY = RGB (0, 162, 232);
    txSetFillColor (SKY);
    txClear ();

    ...
    }
```

Во-третьих, код в абзаце, где рисуется что-то зеленое (фрагмент 2):

```
    txSetColor(TX_GREEN);
    txSetFillColor (TX_GREEN);
    txRectangle(0,400, 800,600);
```

и последние строчки последнего абзаца (фрагмент 5):

```
    txRectangle(545,390,615,490);
    txSetFillColor(WOOD);
    txRectangle(650, 400,720,530);
    txSetColor (TX_BLACK);
    txLine(545,440, 615,440);
    txLine(580,390, 580,490);
```

заметно выбиваются из- за неряшливости.Помните ,мы говорили о типографском форматировании ?Вот тут оно нарушено,и смотрится это откровенно гадко.Скобка без пробела перед ней склеивается с названием команды,первый аргумент команды(цвет или координата)склеивается тоже, и код смотрится в виде куска грязной неаппетитной каши,налипшей на монитор.Фу.Гадко. А вот читать прямо сейчас этот абзац вам не гадко? А тут всего-то добавлено немного лишних пробелов в одних местах и убрано в других. Впечатление, надеюсь, отвратительное. Автор просит прощения у верстальщиков и редакторов этого текста, испытавших культурный шок. Что ж, хороший вкус прививается не только хорошими примерами.

Исправляем:

```
    txSetColor (TX_GREEN);
    txSetFillColor (TX_GREEN);
    txRectangle (0, 400, 800, 600);
```

```
    txSetColor(TX_GREEN);
    txSetFillColor (TX_GREEN);
    txRectangle(0,400, 800,600);
```

```
    txRectangle (545, 390, 615, 490);
    txSetFillColor (WOOD);
    txRectangle (650, 400, 720, 530);
    txSetColor (TX_BLACK);
    txLine (545, 440, 615, 440);
    txLine (580, 390, 580, 490);
```

```
    txRectangle(545,390,615,490);
    txSetFillColor(WOOD);
    txRectangle(650, 400,720,530);
    txSetColor (TX_BLACK);
    txLine(545,440, 615,440);
    txLine(580,390, 580,490);
```

Смотрится отлично. Справа для сравнения приведены прежние, гадкие версии. Видите разницу? Надеемся, да.

Кстати, некоторые подобные погрешности можно поправить автоматически. С помощью функции редактора “Заменить” – в CodeBlocks это команда Replace, Ctrl+R – замените открывающую круглую скобку на пробел и эту же скобку: “(“ на “␣(”. Символом “␣” здесь обозначен пробел. После этого в тексте появятся лишние пробелы, их можно удалить заменой двух пробелов и скобки на пробел и скобку: “␣␣(” на “␣(”, если нужно, несколько раз. Перед пустыми скобками, кстати, пробел часто не ставится, потому что они не содержат аргументов, что может сливаться с названием функции, это тоже можно сделать еще одной заменой. Аналогичным путем добавьте только один пробел после каждой запятой. Возможно, вы об этом не догадываетесь, но мы немного затронули так называемые алгорифмы Маркова, удивительно красивую вещь.

Далее, в-четвертых, фрагмент 3:

```
    txSetColor (TX_GRAY);
    txSetFillColor (TX_GRAY);
    POINT mountains[10] = {{-50, 320}, {50, 150}, {150, 240}, {275, 170}, {400, 270}, {500, 150},
{675, 270}, {800, 120}, {800, 400}, {0, 400}};
    txPolygon (mountains, 10);
```

Здесь строка с определением контура mountains слишком длинна, ее следует разбить на несколько строк. Вот как это сделано не особо аккуратно:

```
       txSetColor (TX_GRAY);
       txSetFillColor (TX_GRAY);
       POINT mountains[10] = {{-50, 320}, {50, 150}, {150, 240}, {275, 170},
{400, 270}, {500, 150}, {675, 270},
{800, 120}, {800, 400}, {0, 400}};
       txPolygon (mountains, 10);
```

А вот как аккуратнее:

```
    txSetColor (TX_GRAY);
    txSetFillColor (TX_GRAY);
    POINT mountains[10] = {{-50, 320}, {50,  150}, {150, 240}, {275, 170}, {400, 270},
                           {500, 150}, {675, 270}, {800, 120}, {800, 400}, {0,   400}};
    txPolygon (mountains, 10);
```

В каждой строке по пять точек, удобно считать. Все точки и все координаты аккуратно выровнены друг под другом, удобно следить за ними. Вот посмотрите, что будет, если сделать опечатку:

```
    txSetColor (TX_GRAY);                                                                  Строка
    txSetFillColor (TX_GRAY);                                                              1
    POINT mountains[10] = {{-50, 320}, {50,  150}, {150, 240}, {275, 170}, {400, 270},    2
                           {500, 150}, {675, 270}, {80, 120}, {800, 400}, {0,   400}};     3
    txPolygon (mountains, 10);                                                             4
                                                                                           5
```

Результатом будет ошибка, которую не найдет компилятор, потому что ему безразлично значение координат. Теперь посмотрим на форматирование: во четвертой строке, начиная с числа 120, все съехало налево на один символ. Тогда либо число 120 должно быть четырехзначным, либо предшествующее ему число 80 – трехзначным. Оп, в числе 80 забыт ноль, на самом деле должно быть 800. А если не держать строгое форматирование, пришлось бы искать кривую вершину по рисунку во время выполнения программы, это гораздо сложнее.

Опытный программист знает, что ошибок избежать невозможно. Еще более опытный умеет перевести ошибки времени исполнения программы (ошибки алгоритма) на стадию компиляции, а еще лучше – до стадии компиляции, на этапе набора текста. Вот это тот самый случай.

В пятых, во фрагменте 4, где рисуется дом, есть две проблемы. Первая:

```
    POINT a[3] ={{510,320},{635,190}, {760,320}};
    txPolygon (a, 3);
```

Имя контура "a" непонятно что означает. Может быть, это ускорение в физике? Ускорение чего, дома? Он движется? Или это коэффициент квадратного уравнения $ax^2 + bx + c = 0$, которое который год уже неусыпно решают его жители? ХЗ, что, конечно, означает "хочется задуматься" – над именем "a". Ясность программы явно нарушена. Опытным путем устанавливаем, за что отвечает это "a" – убираем эти строки, и, вуаля, у дома пропадает крыша с чердаком. О! Значит, "a" – это сокращение от чердака – attic! Срочно переименовываем, заодно расставляем пробелы, как нужно (вам тоже было противно смотреть на эту строку?):

```
    POINT attic[3] = {{510, 320}, {635, 190}, {760, 320}};
    txPolygon (attic, 3);
```

И становится гораздо понятнее. В этом сила хороших имен. Можно было, кстати, назвать более известным словом – roof, крыша. Уж у нее точно не будет ускорения "a", хотя иногда, говорят, она у некоторых съезжает, а если с ускорением, то вообще кошмар.

Ладно. Мы за психическое здоровье, оно программистам очень нужно.

Далее, в-шестых. Чуть ниже во фрагменте 4 определяется константа WOOD с помощью конструкции

```
    const COLORREF WOOD = RGB (103, 65, 44);
```

которая затем позволяет использовать имя WOOD для указания цвета двери. (Прочитайте про функцию RGB в системе помощи TXLib.) Это делается для того, чтобы не использовать цветовые компоненты (103, 65, 44) постоянно, и способствует пониманию кода. Но это определение "въехало" в середину фрагмента, рисующего дом, и появляется там неожиданно. Лучше такие понятные названия вынести в отдельный фрагмент, можно даже в начало программы. Можно даже сразу после раздела подключения библиотек, тогда при чтении программы не будет неожиданных определений.

Получается так:

```
  const COLORREF HOUSE = RGB (185, 122, 87);
  const COLORREF WOOD  = RGB (103,  65, 44);

  txSetColor     (HOUSE);
  txSetFillColor (HOUSE);
  txRectangle (510, 320, 760, 540);
  POINT attic[3] = {{510, 320}, {635, 190}, {760,
320}};
  txPolygon (attic, 3);

  txSetColor     (WOOD);
  txSetFillColor (TX_GRAY);
  txRectangle (545, 390, 615, 490);

  txSetFillColor (WOOD);
  txRectangle (650, 400, 720, 530);

  txSetColor (TX_BLACK);
  txLine (545, 440, 615, 440);
  txLine (580, 390, 580, 490);
```

```
  const COLORREF HOUSE = RGB (185, 122, 87);
  txSetColor (HOUSE);
  txSetFillColor (HOUSE);
  txRectangle (510, 320, 760, 540);
  POINT a[3] ={{510,320},{635,190},
{760,320}};
  txPolygon (a, 3);
  const COLORREF WOOD = RGB (103, 65, 44);
  txSetColor (WOOD);
  txSetFillColor (TX_GRAY);
  txRectangle(545,390,615,490);
  txSetFillColor(WOOD);
  txRectangle(650, 400,720,530);
  txSetColor (TX_BLACK);
  txLine(545,440, 615,440);
  txLine(580,390, 580,490);
```

Справа – старый код. Здесь дополнительно добавлены несколько пустых строк и выровнены названия цветов друг под другом, чтобы было легче за ними следить. Хотя если от этого в коде теряются границы между объектами, то так не делайте. Чуть позже мы разобьем программу на части несколько другим образом, и дополнительные пустые строки перестанут мешать.

Шесть проблем это немного, их легко исправить. В итоге все получается так (справа – старый вариант):

```
#include "TXLib.h"

const COLORREF SKY  = RGB (0,   162, 232);
const COLORREF HOUSE = RGB (185, 122, 87);
const COLORREF WOOD  = RGB (103, 65,  44);

int main()
  {
  txCreateWindow (800, 600);

  txSetFillColor (SKY);
  txClear();

  txSetColor(TX_GREEN);
  txSetFillColor (TX_GREEN);
  txRectangle (0, 400, 800, 600);

  txSetColor (TX_GRAY);
  txSetFillColor (TX_GRAY);
  POINT mountains[10] = {{-50, 320}, {50,  150},
                         {150, 240}, {275, 170},
                         {400, 270}, {500, 150},
                         {675, 270}, {800, 120},
                         {800, 400}, {0,   400}};
  txPolygon (mountains, 10);

  txSetColor (HOUSE);
  txSetFillColor (HOUSE);
  txRectangle (510, 320, 760, 540);
  POINT roof[3] = {{510, 320}, {635, 190}, {760,
320}};
  txPolygon (roof, 3);

  txSetColor     (WOOD);
  txSetFillColor (TX_GRAY);
  txRectangle (545, 390, 615, 490);

  txSetFillColor (WOOD);
  txRectangle (650, 400, 720, 530);

  txSetColor (TX_BLACK);
  txLine (545, 440, 615, 440);
  txLine (580, 390, 580, 490);
  }
```

```
#include "TXLib.h"

int main()
 {

 txCreateWindow (800, 600);

 const COLORREF SKY = RGB (0, 162, 232);
 txSetFillColor (SKY);
 txClear ();

 txSetColor(TX_GREEN);
 txSetFillColor (TX_GREEN);
 txRectangle(0,400, 800,600);

 txSetColor (TX_GRAY);
 txSetFillColor (TX_GRAY);
 POINT mountains[10] = {{-50, 320}, {50,
150}, {150, 240}, {275, 170}, {400, 270},
{500, 150}, {675, 270}, {800, 120}, {800,
400}, {0, 400}};
 txPolygon (mountains, 10);

 const COLORREF HOUSE = RGB (185, 122, 87);
 txSetColor (HOUSE);
 txSetFillColor (HOUSE);
 txRectangle (510, 320, 760, 540);
 POINT a[3] ={{510,320},{635,190},
{760,320}};
 txPolygon (a, 3);
 const COLORREF WOOD = RGB (103, 65, 44);
 txSetColor (WOOD);
 txSetFillColor (TX_GRAY);
 txRectangle(545,390,615,490);
 txSetFillColor(WOOD);
 txRectangle(650, 400,720,530);
 txSetColor (TX_BLACK);
 txLine(545,440, 615,440);
 txLine(580,390, 580,490);
 }
```

Ну что, лучше? Лучше! Ты – не ты, когда твой код неаккуратен. Можно передохнуть и съесть Snickers.

Теперь посмотрите на свой код новыми глазами и исправьте в нем то, что вам перестало нравиться. Надеюсь, вы будете более строги к себе, чем автор в своем ревью. Трудно? Поначалу трудно. *Тяжела и неказиста жизнь системного программиста.* Но со временем придет привычка быстро замечать такие вещи и устранять их уже на этапе набора текста программы. Если вы при наборе уже нормально все форматируете, ревью будет проще.

Но оно все равно будет, потому что нет предела совершенству, и никакое совершенство не достигается сразу.

Почему жестокий автор не сказал все это перед заданием? Вы должны были получить свои личные ощущения, свой личный опыт, на вашем примере, на ваших героях, каждая строчка которых была вами выстрадана. Загуглите грузинскую притчу "Заработанный рубль". Этот ваш личный опыт ничто не заменит. Не избегайте его, он – фундамент вашего профессионализма.

1.2. Функции

Итак, вы справились с первым заданием – написали свою первую графическую программу на Си. Позади десятки минут рисования, измерения, набора текста и отладки. Первое задание часто получается сложным, потому что в него включается еще и установка и настройка системы программирования, да и вообще много непривычных действий. Но вы справились.

Давайте подумаем над вопросом в конце предыдущей главы – тем, что про утомительные действия. Почему это важно? "Безусловный рефлекс" любого программиста – анализировать и уменьшать, сводить до нуля такие проблемы. Это часто называют "принципом разумной лени" – не делать вручную то, что можно переложить на компьютер, алгоритм, язык программирования, создать технологию, убирающую ручной труд, механическое запоминание, перегрузку головы информацией. Поэтому программисты часто очень рациональны и эффективны, и не только в программировании.

Скорее всего, вам очень не понравилось измерять тонну координат точек. Это действительно неприятно, и мы этим займемся в следующей главе. Сейчас заметим, что использование MS Paint вместо бумаги удобнее тем, что Paint показывает координаты мышки, а бумага нет. Так, кстати, делает не каждый графический редактор.

Но наверняка вы начали потихоньку *путаться в коде.* Да, вы разделили текст на абзацы пустыми строками (ведь разделили же? Иначе вообще жесть). Но вот за что отвечает третий абзац? За рисование дерева или дома, к примеру? А предпоследний? Рисует человека или котика? Неясно. Можно, конечно, убрать какой-то абзац из программы и посмотреть, какой объект исчезнет. Но это неудобно, согласитесь.

Далее вот еще такое рассуждение. Давайте мысленно увеличим количество героев и, соответственно, объем программы. Такой процесс – мысленное увеличение какого-либо параметра в программе – называется *масштабированием,* оно очень важно в разработке программной архитектуры. Уверен, что вы этим уже занимались до программирования. (Однажды автор в детстве, не промасштабировав свою любовь к сгущенке, поспорил, что съест ее в один присест целый килограмм. С тех пор автор не очень любит сгущенку :) и понимает важность масштабирования.)

Так вот, при масштабировании проблема путаницы резко увеличивается. А серьезные программы гораздо больше нашей. Поэтому, если мы хотим научиться их писать, надо срочно придумать какое-то средство, уменьшающее эту путаницу.

Давайте подумаем на более знакомом материале. Представьте себе тонну сплошного текста, разделенную только на абзацы. Что бы вы с ней сделали, чтобы стала понятной, чтобы она приобрела *структуру?* Думаю, вы озаглавите каждый абзац, и по этим заголовкам будете отличать их друг от друга. Человеческое мышление вообще во многом работает через *именование,* кстати.

В языке Си есть такая возможность, она называется *созданием функции.* Функция – это часть программы с собственным именем. Раз ее определив, ей можно легко пользоваться в других местах программы. Функцию сделать очень легко:

1) Мы выносим один из абзацев текста из функции main() ниже ее, за пределы закрывающей фигурной скобки. (Выделяем текст, Ctrl+X, идем в конец программы, Ctrl+V.)
2) Окружаем вынесенный текст собственными фигурными скобками.

3) Перед открывающей фигурной скобкой пишем *заголовок функции* - ее имя, по аналогии с функцией main. Только, конечно, не надо ее называть main – такая функция уже есть. Не надо и main2 – это неясно совсем. Не надо и сильно кратко – cat или man, например, хотя это уже лучше. Лучше всего спросить себя, что функция делает – и ответить словосочетанием. Например, если она рисует котика, то пусть называется DrawCat. Или RisuiKotika (это называется транслит). Или PUCYU_KOTUKA, это другой тип транслита. Или DrawMan, если она рисует человечка. Лучше, конечно, по-английски, так повсеместно принято в программировании. Но автору встречались ученики, называющие функции по-исландски. :)

4) Мы получили *определение функции.* Оно состоит из заголовка, содержащего ее имя, и *тела* – кода, заключенного в фигурные скобки.

```
#include "TXLib.h"

int main()
    {
    txCreateWindow (800, 600);

    txSetColor (TX_WHITE);
    txSetFillColor (TX_TRANSPARENT);
    txRectangle (10, 10, 790, 590);

    txSetColor (TX_LIGHTCYAN);
    txEllipse (200, 150, 600, 450);
    txEllipse (245, 150, 555, 450);
    txEllipse (290, 150, 510, 450);
    txEllipse (330, 150, 470, 450);
    txEllipse (375, 150, 425, 450);
    txEllipse (200, 150, 600, 450);
    txEllipse (200, 190, 600, 410);
    txEllipse (200, 230, 600, 370);
    txEllipse (200, 270, 600, 330);
    txLine    (200, 300, 600, 300);

    txSetColor (TX_LIGHTGREEN);
    txSelectFont ("Times New Roman", 60);
    txSetTextAlign (TA_CENTER);
    txTextOut (400, 480, "Hello, world!");

    //-----------------------------------------------//
    // (Отсюда мы вырезали код, рисующий человечка)  //
```

```
    //-----------------------------------------------//

    txSetFillColor (TX_TRANSPARENT);
    txLine (400, 75, 400, 150);
    txRectangle (400, 75, 450, 115);
    txSelectFont ("Times New Roman", 20);
    txTextOut (425, 85, "C++");

    txTextCursor (false);
    return 0;
    }

void DrawMan()                          // Заголовок функции   \
    {                                   // \                    |
    txSetColor (TX_YELLOW);              //  |                   |
    txSetFillColor (TX_YELLOW);          //  |                   |
    txLine   (385, 135, 385, 120);       //  |                   |
    txLine   (385, 135, 375, 150);       //   > Тело функции     > Определение функции
    txLine   (385, 135, 395, 150);       //  |    DrawMan        |       DrawMan
    txLine   (385, 125, 375, 135);       //  |                   |
    txLine   (385, 125, 400, 120);       //  |                   |
    txCircle (385, 115, 6);              //  |                   |
    }                                    // /                   /
```

Далее:

5) Помещаем перед функцией main копию заголовка функции с точкой с запятой в конце – это называется *прототип функции.* Он дает информацию компилятору о том, что такая функция в программе будет определена. (Помним, что компилятор – программа, переводящая текст на Си в машинные коды для исполнения.) Без этого функцией нельзя будет полноценно пользоваться. Дело в том, что компилятор Си транслирует код программы сверху вниз и не может заранее заглянуть в конец программы, где находится определение функции.

6) В то место, откуда мы вырезали код, рисующий героя, вписываем *вызов функции* – ее имя, круглые скобки и точку с запятой. Посмотрите, что получилось:

```
#include "TXLib.h"

void DrawMan();                         // Прототип функции DrawMan

int main()
    {
    txCreateWindow (800, 600);

    txSetColor (TX_WHITE);
    txSetFillColor (TX_TRANSPARENT);
    txRectangle (10, 10, 790, 590);

    txSetColor (TX_LIGHTCYAN);
    txEllipse (200, 150, 600, 450);
    txEllipse (245, 150, 555, 450);
    txEllipse (290, 150, 510, 450);
    txEllipse (330, 150, 470, 450);
    txEllipse (375, 150, 425, 450);
    txEllipse (200, 150, 600, 450);
    txEllipse (200, 190, 600, 410);
    txEllipse (200, 230, 600, 370);
    txEllipse (200, 270, 600, 330);
    txLine    (200, 300, 600, 300);
```

```
    txSetColor (TX_LIGHTGREEN);
    txSelectFont ("Times New Roman", 60);
    txSetTextAlign (TA_CENTER);
    txTextOut (400, 480, "Hello, world!");

    DrawMan();                          // Вызов функции DrawMan

    txSetFillColor (TX_TRANSPARENT);
    txLine (400, 75, 400, 150);
    txRectangle (400, 75, 450, 115);
    txSelectFont ("Times New Roman", 20);
    txTextOut (425, 85, "C++");

    txTextCursor (false);
    return 0;
    }

void DrawMan()                          // Заголовок функции   \
    {                                   // \                    |
    txSetColor (TX_YELLOW);              //  |                   |
    txSetFillColor (TX_YELLOW);          //  |                   |
    txLine   (385, 135, 385, 120);       //  |                   |
    txLine   (385, 135, 375, 150);       //   > Тело функции     > Определение функции
    txLine   (385, 135, 395, 150);       //  |    DrawMan        |       DrawMan
    txLine   (385, 125, 375, 135);       //  |                   |
    txLine   (385, 125, 400, 120);       //  |                   |
    txCircle (385, 115, 6);              //  |                   |
    }                                    // /                   /
```

При исполнении процессор компьютера сначала последовательно выполнит строчки из функции main, пока не дойдет до вызова функции DrawMan. Ее вызов приведет к тому, что процессор "прыгнет" на первую строчку тела функции DrawMan, и также последовательно выполнит все строчки из нее. Дойдя до конца определения, процессор "прыгнет" назад, обратно в main (это называется *возврат из функции*), и продолжит исполнять команды из main, пока не дойдет до ее конца.

Все! Смотрите, по коду, связанному с рисованием человечка, вопросов нет – с ним все ясно. Одна строчка в главной функции main для вызова – вместо душного абзаца из команд TXLib'a. Хм, понятно ли, что мы дальше делаем? Конечно, срочно делим весь main на части:

```
//---------------------------------------------------------------------------

#include "TXLib.h"

//---------------------------------------------------------------------------

void DrawMan();
void DrawEarth();
void DrawFlag();
void DrawHello();
void DrawFrame();

//---------------------------------------------------------------------------

int main()
    {
    txCreateWindow (800, 600);

    DrawFrame();
    DrawEarth();
```

```
    DrawHello();
    DrawMan();
    DrawFlag();

    txTextCursor (false);
    return 0;
    }

//---------------------------------------------------------------------------

void DrawEarth()
    {
    txSetColor (TX_LIGHTCYAN);

    txEllipse (200, 150, 600, 450);
    txEllipse (245, 150, 555, 450);
    txEllipse (290, 150, 510, 450);
    txEllipse (330, 150, 470, 450);
    txEllipse (375, 150, 425, 450);
    txEllipse (200, 150, 600, 450);
    txEllipse (200, 190, 600, 410);
    txEllipse (200, 230, 600, 370);
    txEllipse (200, 270, 600, 330);
    txLine    (200, 300, 600, 300);
    }

//---------------------------------------------------------------------------

void DrawMan()
    {
    txSetColor     (TX_YELLOW);
    txSetFillColor (TX_YELLOW);

    txLine   (385, 135, 385, 120);
    txLine   (385, 135, 375, 150);
    txLine   (385, 135, 395, 150);
    txLine   (385, 125, 375, 135);
    txLine   (385, 125, 400, 120);
    txCircle (385, 115, 6);
    }

//---------------------------------------------------------------------------

void DrawFlag()
    {
    txSetFillColor (TX_TRANSPARENT);

    txLine (400, 75, 400, 150);
    txRectangle (400, 75, 450, 115);
    txSelectFont ("Times New Roman", 20);
    txTextOut (425, 85, "C++");
    }

//---------------------------------------------------------------------------

void DrawHello()
    {
    txSetColor (TX_LIGHTGREEN);
    txSelectFont ("Times New Roman", 60);

    txSetTextAlign (TA_CENTER);
    txTextOut (400, 480, "Hello, world!");
    }

//---------------------------------------------------------------------------
```

```
void DrawFrame()
    {
    txSetColor (TX_WHITE);
    txSetFillColor (TX_TRANSPARENT);

    txRectangle (10, 10, 790, 590);
    }
```

Супер! Все ясно!

Здесь еще дополнительно вставлены строки, начинающиеся с двух знаков деления, или слешей "//", после которых идут тире, сливающиеся в горизонтальные линии. Эти строки визуально отделяют функции друг от друга, подчеркивая структуру программы. Они не воспринимаются компилятором и не мешают ему обрабатывать текст программы. Если вы работаете в сборке CodeBlocks, скачанной с сайта ded32.net.ru, их можно легко вставить, набрав в тексте букву "j" (или "J") и сразу после этого нажав "Ctrl+J". Крайне рекомендуется, чтобы вы пользовались такими разделителями.

Заметим еще, что функциями профессиональные программисты пользуются постоянно, и поэтому вам не придется переучиваться в дальнейшем. Это вообще проявление самого основополагающего принципа в программировании – "разделяй и властвуй". Так что римские императоры, пожалуй, тоже были своего рода программистами. :)

В каком порядке располагать прототипы и определения функций? Формально (но только лишь формально) это неважно: если заранее дать компилятору прототип, то он найдет определение где угодно в файле. Но лучше располагать функции по степени их важности для проекта. Функции, логически связанные друг с другом, лучше располагать вместе, начиная от более "главной", переходя к "второстепенным". Поэтому наиболее грамотное расположение – главная функция (main) сверху, далее более второстепенные, которые вызываются из main. Под каждой второстепенной функцией – снова те, что из нее вызываются и так далее. Это позволяет быстрее читать код, не "прыгать" по файлу в поисках определений функций. Для более сложных программ применяется послойный метод – сверху группа функций верхнего уровня, далее группы родственных вспомогательных функций и т.д. Главное, чтобы была внятная логичная структура, а не просто "я пишу вот так, мне так удобно, работает и ладно". В частности, устаревший стиль порядка функций, взятый из древнего Паскаля, когда main пишется снизу, иногда дается на уроках, кружках и курсах. Не надо следовать таким упрощенным устаревшим стилям. Лучше задумайтесь о смене курсов.

Заметим, что то, что мы ранее называли "командами TXLib" (txLine, txCircle и т.п.) на самом деле тоже функции. Их определение лежит в файле TXLib.h, который вы подключаете к своей программе в самом ее начале. В этот файл можно заглянуть и посмотреть определения этих функций. Например, для функций txCircle, txEllipse и txRectangle они достаточно простые, там вызываются аналогичные системные функции, которые и рисуют фигуры.

Если вы человек внимательный, то увидели, что в начале определения функции и в прототипе стоит слово "void". Что это такое? Дело в том, что функция может нести математический смысл, то есть вычислять что-либо, например, по какой-то формуле. Тогда у функции будет некоторый численный результат, который она вернет как итог своих вычислений. В наших функциях рисования нет никаких формул и математического результата, поэтому перед функциями мы это явно указываем - пишем слово "void" – "пустой", "отсутствующий" (в математическом смысле). Разумеется, можно писать и функции с возвратом результата, с этим мы познакомимся далее.

1.2.1. Рефакторинг кода

Что мы с вами только что сделали? Мы изменили, улучшили свой код, переписав его. Уже во второй раз, до этого мы переписали код после ревью первого задания, до функций. В программировании для такого переписывания, улучшения используют термин *рефакторинг.* Это очень важная вещь. Во время рефакторинга улучшается структура, архитектура программы, она становится ясной, легко читаемой, легко масштабируемой. Да, на это тратится время. Но оно окупается. Конечно, не стоит бесконечно проводить рефакторинг, не приближаясь к результату. Надо сохранять баланс.

Зачем программистам было нужно придумывать такое мудреное слово? Так обстояли дела, что даже профессионалы не придавали большого значения этому процессу, в результате код становился "грязным", непонятным, плохо масштабируемым, появлялись сложные ошибки. Автору этого подхода, Мартину Фаулеру, пришлось написать целую книгу об этом, и это очень сильно повлияло на индустрию программирования.

Тут надо сказать, что программисты прям не то чтобы все подряд ленивые и пишут плохой код. Это не так, каждый очень старается. Но не всегда заранее понятно, как лучше писать ту или иную часть программы. Часто приходится писать некоторый начальный вариант кода, после чего становится ясно, как сделать лучше. И приходится рефакторить.

Термин "рефакторинг" может помочь вам не только при написании программ. Вы сделали не лучшую презентацию и вам неохота ее *переделывать?* Просто сделайте *рефакторинг* – это же куда более крутой процесс. ;) Вы плохо написали контрольную и пошли ее переписывать? Скажите дома, что у вас *рефакторинг контрольной,* и вас не то что не будут ругать дома, а даже станут больше уважать. :) Терминология – великая штука!

1.2.2. Пример, который снова не надо копировать

Вы поняли, почему. Выше была глава с таким же названием – прочтите ее. Все, что там написано, относится и к этому примеру.

Этот пример входит в состав примеров TXLib (Пример: Функции).

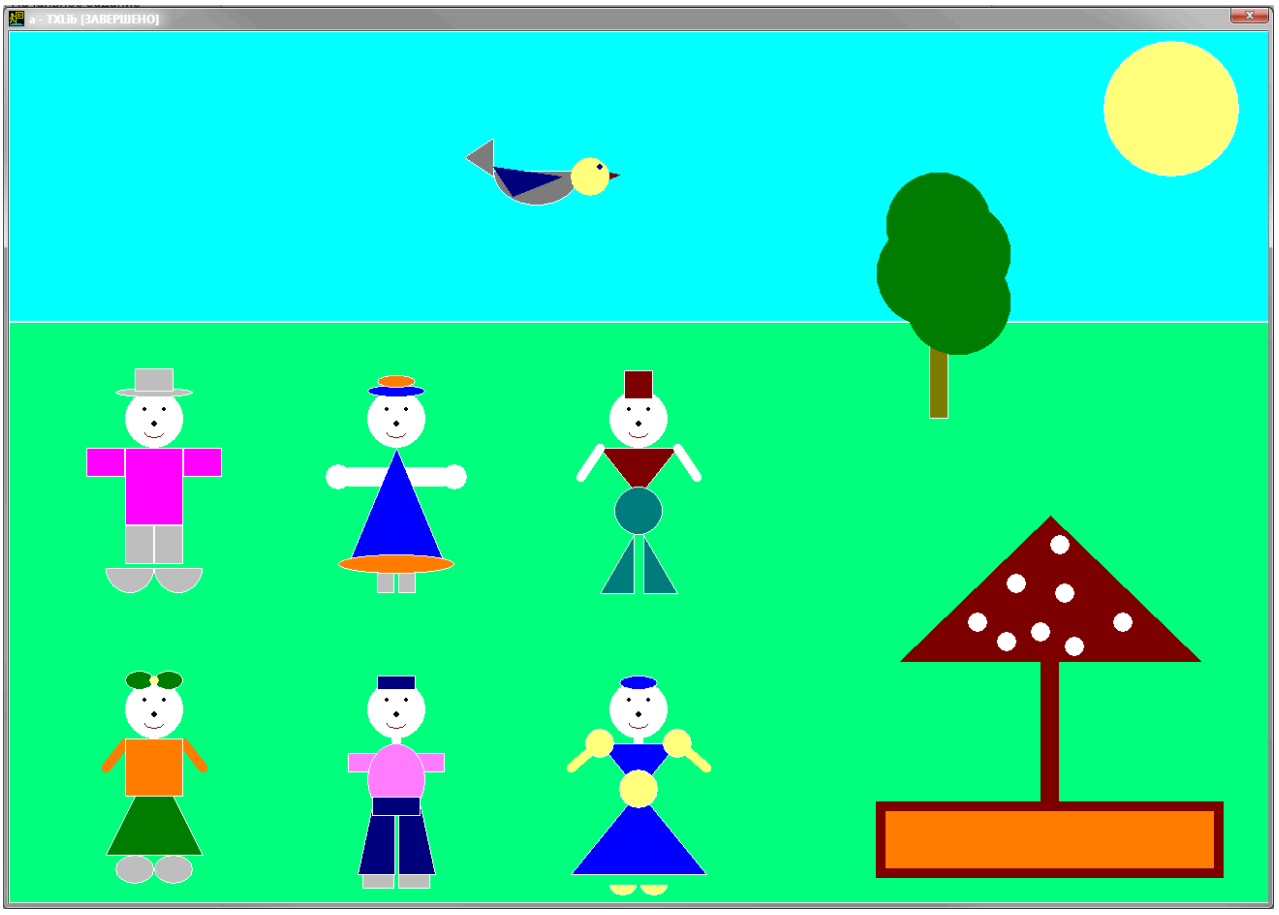
```
#include "TXLib.h"

//------------------------------------------------------------------------

void DrawMan();
void DrawEarth();
void DrawFlag();
void DrawHello();
void DrawFrame();

//------------------------------------------------------------------------

int main()
    {
    txCreateWindow (800, 600);

    DrawFrame();
    DrawEarth();
    DrawHello();
```

```
    DrawMan();
    DrawFlag();

    txTextCursor (false);
    return 0;
    }

//---------------------------------------------------------------------------

void DrawEarth()
    {
    txSetColor (TX_LIGHTCYAN);
    txEllipse (200, 150, 600, 450);
    txEllipse (245, 150, 555, 450);
    txEllipse (290, 150, 510, 450);
    txEllipse (330, 150, 470, 450);
    txEllipse (375, 150, 425, 450);
    txEllipse (200, 150, 600, 450);
    txEllipse (200, 190, 600, 410);
    txEllipse (200, 230, 600, 370);
    txEllipse (200, 270, 600, 330);
    txLine    (200, 300, 600, 300);
    }

//---------------------------------------------------------------------------

void DrawMan()
    {
    txSetColor (TX_YELLOW);
    txSetFillColor (TX_YELLOW);
    txLine   (385, 135, 385, 120);
    txLine   (385, 135, 375, 150);
    txLine   (385, 135, 395, 150);
    txLine   (385, 125, 375, 135);
    txLine   (385, 125, 400, 120);
    txCircle (385, 115, 6);
    }

//---------------------------------------------------------------------------

void DrawFlag()
    {
    txSetFillColor (TX_TRANSPARENT);
    txLine (400, 75, 400, 150);
    txRectangle (400, 75, 450, 115);
    txSelectFont ("Times New Roman", 20);
    txTextOut (425, 85, "C++");
    }

//---------------------------------------------------------------------------

void DrawHello()
    {
    txSetColor (TX_LIGHTGREEN);
    txSelectFont ("Times New Roman", 60);
    txSetTextAlign (TA_CENTER);
```

```
    txTextOut (400, 480, "Hello, world!");
    }

//---------------------------------------------------------------------------

void DrawFrame()
    {
    txSetColor (TX_WHITE);
    txSetFillColor (TX_TRANSPARENT);
    txRectangle (10, 10, 790, 590);
    }
```

1.2.3. Ревью кода

Пусть некто написал программу, дающую такую картинку:



Красиво. Вот код, он приведен в сокращении, символ многоточия “...” означает. что из кода для краткости был убран фрагмент.

```
#include "TXLib.h"

    void DrawFon();
    void DrawBoy1();
    void DrawGirl1();
    void DrawBoy2();
    void DrawGirl2();
    void DrawBoy3();
    void DrawGirl3();
    void DrawBird();
```

```
int main ()
  {
     txCreateWindow (1300, 900);
     DrawFon();
     DrawBoy1();
     DrawGirl1();
     DrawBoy2();
     DrawGirl2();
     DrawBoy3();
     DrawGirl3();
     DrawBird();
     return 0;
  }
//============================================================================

void DrawBoy1()
  {
     txSetFillColor (TX_WHITE);
     txSetColor (TX_WHITE, 1);
     txCircle (150, 400, 30);
     txSetFillColor (TX_BLACK);
     txCircle (160, 390, 3);
     txCircle (140, 390, 3);
     txCircle (150, 405, 4);
     txSetColor (TX_RED, 1);
     txArc (140, 410, 160, 420, 180, 180);
     txSetColor (TX_WHITE, 1);
     ...
     txSetFillColor (TX_LIGHTGRAY);
     txEllipse (110, 368, 190, 378);
     txRectangle (130, 348, 170, 372);
     txRectangle (120, 510, 150, 550);
     txRectangle (150, 510, 180, 550);
     txChord (100, 530, 150, 580, 180, 180);
     txChord (150, 530, 200, 580, 180, 180);
  }

//============================================================================

void DrawGirl1()
  {
     txSetFillColor (TX_WHITE);
     txCircle ( 400, 400, 30);
     txSetFillColor (TX_BLACK);
     txCircle (410, 390, 3);
     txCircle (390, 390, 3);
     txCircle (400, 405, 4);
     ...
  }

//============================================================================

void DrawBoy2()
  {
```

```
      txSetFillColor (TX_WHITE);
      txCircle ( 650, 400, 30);
      txSetFillColor (TX_BLACK);
      txCircle (660, 390, 3);
      txCircle (640, 390, 3);
      txCircle (650, 405, 4);
      ...
    }

//============================================================================

  void DrawGirl2()
    {
      txSetColor (TX_WHITE, 1);
      txSetFillColor (TX_WHITE);
      txCircle ( 150, 700, 30);
      txSetFillColor (TX_BLACK);
      txCircle (160, 690, 3);
      txCircle (140, 690, 3);
      txCircle (150, 705, 4);
      ...
    }
//============================================================================

  void DrawBoy3()
    {
      txSetFillColor (TX_WHITE);
      txCircle ( 400, 700, 30);
      txRectangle (395, 730, 405, 735);
      txSetFillColor (TX_BLACK);
      txCircle (410, 690, 3);
      txCircle (390, 690, 3);
      txCircle (400, 705, 4);
      ...
    }

//============================================================================

  void DrawGirl3()
    {
      txSetFillColor (TX_WHITE);
      txCircle ( 650, 700, 30);
      txRectangle (645, 730, 655, 735);
      txSetFillColor (TX_BLACK);
      txCircle (660, 690, 3);
      txCircle (640, 690, 3);
      txCircle (650, 705, 4);
      txSetColor (TX_RED, 1);
      txArc (640, 710, 660, 720, 180, 180);
      txSetColor (TX_WHITE, 1);
      txSetFillColor (TX_LIGHTBLUE);
      POINT star9[4] = {{610, 735}, {690, 735}, {580, 870}, {720, 870}};
      txPolygon (star9, 4);
      txEllipse (630, 665, 670, 680);
      txSetFillColor (TX_YELLOW);
      txCircle ( 610, 735, 15);
```

```
      txCircle ( 690, 735, 15);
      txCircle ( 650, 782, 20);
      txSetColor (TX_YELLOW, 10);
      txLine (610, 734, 580, 760);
      txSetColor (TX_YELLOW, 10);
      txLine (690, 734, 720, 760);
      txSetColor (TX_WHITE, 1);
      txSetFillColor (TX_YELLOW);
      txChord (620, 872, 648, 892, 180, 180);
      txChord (652, 872, 680, 892, 180, 180);
    }

//============================================================================

  void DrawFon()
    {
      txSetFillColor (TX_LIGHTCYAN);
      txRectangle (0, 0, 1400, 300);
      txSetFillColor (TX_LIGHTGREEN);
      ...

      txCircle (1085, 530, 10);
      txCircle (1040, 570, 10);
      txCircle (1090, 580, 10);
      txCircle (1065, 620, 10);
      ...
    }

//============================================================================

  void DrawBird()
    {
      txSetFillColor (TX_DARKGRAY);
      txPie (590, 110, 500, 180, 180, 180);
      POINT star14[3] = {{500, 150}, {470, 130}, {500, 110}};
      txPolygon (star14, 3);
      ...
    }
```

Так. Тоже вроде бы неплохо, человек явно старался. Но мы знаем, что нет предела совершенству, поэтому сделаем ревью.

Во-первых, здесь довольно странная система отступов, они в этом коде тройные. Все функции, вместе с их заголовками, отступлены на два пробела, далее скобки еще на два пробела, а тела определений отступлены дополнительно еще на три пробела. Это создает визуальную путаницу и поэтому неудобно. Однако автор педантично придерживается этого стиля. Что ж, его можно за это уважать, и, пока он его не нарушит, это его право. Но если он начнет запутываться, или хоть один раз отступит от своих правил, то все – рефакторинг неизбежен. Мы бы советовали автору, пока он не привык сильно к своим странным тройным отступам, убрать их, оставив отступы обычные, одинарные. Дело в том, что при дальнейшем развитии программы структура отступов и так усложнится, и дополнительные отступы точно начнут мешать.

Кстати, автор кода тут уже за собой не уследил: после определений функций main и DrawGirl2 почему-то нет пустых строк, а после определений остальных функций – есть. Значит, правило

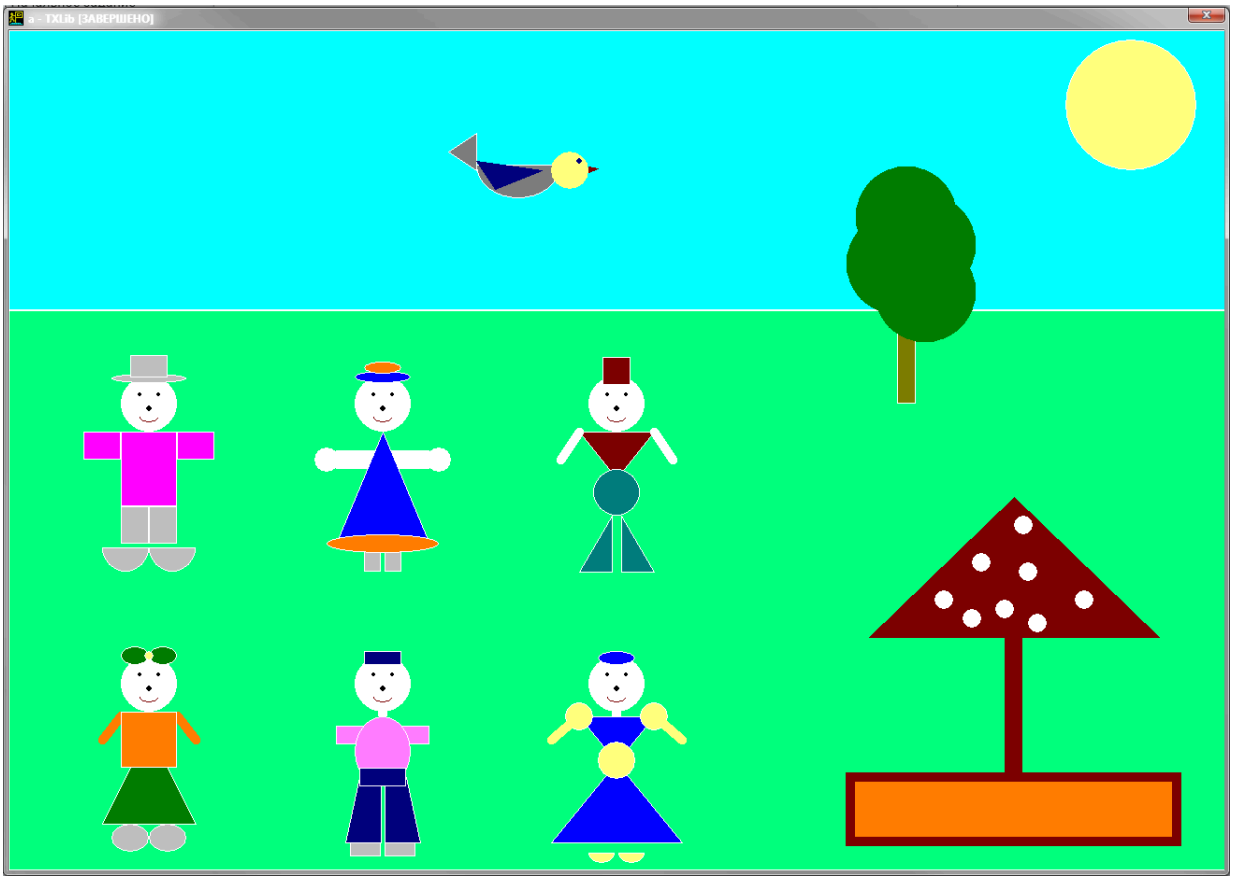
стиля нарушено, и автору уже стоит переформатировать текст, включая отступы. В будущем ему будет от этого сильно проще.

Во-вторых, здесь довольно сложные объекты, и код внутри функций следовало бы разделить на части пустыми строками для большей структурности. Это надо сделать не откладывая, пока автор помнит, за какую часть каждого объекта отвечает каждая строка кода. Потом это забудется и разделить на части функции уже не получится. При дальнейшей работе с объектами это вызовет трудности.

Третья проблема здесь в именах. Они не особо информативны, в том смысле. что плохо отделяют объекты друг от друга. Имена всех этих мальчиков Boy1, Boy2, Boy3 и девочек Girl1, Girl2, Girl3 сливаются друг с другом, как в большой тусовке. Да и обидятся девочки, если их так называть. Да и мальчики тоже (но, может быть, гордо этого не покажут).

В функции DrawGirl3 также есть имя, обозначающее контур многоугольника, но оно почему-то называется star9, хотя никакой звезды у женщины в синем нет. Фон этого многоугольника – синий. Это платье. В этом нетрудно убедиться, если убрать строку с рисованием этого многоугольника. Какая связь название star9 с платьем – загадка, автор предполагает, что дело в копировании имени из примера к функции txPolygon, где рисуется кривая звездочка. В какой-то момент к переменной star из этого примера прилепилась цифра 9. Она каждый раз будет вызывать вопрос, почему именно 9, где star8 и star10 (а их нет), и это будет съедать наше внимание, которого никогда не хватает.

В дальнейшей работе с кодом это создаст трудности, когда мы начнем писать функции с движением и взаимодействием объектов, и тогда подобные номера будут еще сильнее путаться. Вообще, использование номеров или цифр в именах – плохой признак, это означает, что человек относится к коду либо слишком формально, либо не напрягает должным образом фантазию, добиваясь ясности текста. Как бы героев переименовать? Посмотрим еще раз на картинку:



Лучше всего здесь, конечно, исходить из сценария будущего мультфильма. Он такой: папа (розовый человечек в шляпе сверху слева) и мама (справа от него) привели троих детей, старшего – крутого качка Славу (сверху), и младших, Дашу и Ваню (снизу), играть в песочницу. Слава, понятное дело, будет следить за мелкими. Последняя тетенька снизу – это орнитолог, изучающая летающую выше Птицу. Далее по сюжету в песочницу приползет радиоактивный Паук, все его испугаются, включая и крутого качка Славу, но умная тетенька орнитолог особым образом призовет Птицу, которая немедленно сожрет страшного Паука, избавив детей от непростой судьбы Питера Паркера. Это только в кино хорошо быть человеком-пауком, да и там ему часто приходилось трудно. Мультфильм заканчивается рассказом о пользе птиц, призывом учить орнитологию, да и вообще посещать уроки биологии, а не прогуливать их в качалке (последнее относится к Славе). Такая вот социальная реклама школьных предметов.

Исходя из этого, героев можно переименовать:

```
Boy1  –> Father
Girl1 –> Mother
Boy2  –> KachokSlava (ему нравится, когда его называют качком)
Girl2 –> Dasha
Boy3  –> Vanya
Girl3 –> ValerianaVarfolomeevnaTheOrnitologist (ну пусть будет так)
```

Заодно функцию DrawFon переименуем в DrawBackground или DrawBack, так как Fon – это не "фон" по-английски. Конечно, мы немного отходим от этого языка, например, назвав Славу Kachk'ом, а не Jock'ом. Но тут уж постирония такая случилась.

Сценарий рисунка – функция main – тогда будет выглядеть так (как всегда, справа – прежний вариант):

```
int main ()
    {
    txCreateWindow (1300, 900);

    DrawBack();

    DrawFather();
    DrawMother();
    DrawKachokSlava();

    DrawDasha();
    DrawVanya();
    DrawValerianaVarfolomeevnaTheOrnitologist();

    DrawBird();

    return 0;
    }
```

```
int main ()
  {
    txCreateWindow (1300, 900);
    DrawFon();
    DrawBoy1();
    DrawGirl1();
    DrawBoy2();
    DrawGirl2();
    DrawBoy3();
    DrawGirl3();
    DrawBird();
    return 0;
  }
```

Функция дополнительно разбита пустыми строками на смысловые части – создание окна, рисование фона, рисование верхнего ряда людей, рисование нижнего ряда, рисование птицы и оператор возврата значения в операционную систему. Код стал яснее и понятнее, с ним будет легко работать в дальнейшем.

Функция DrawGirl3 тоже претерпела изменения. У нее изменено имя, убраны лишние аргументы функции txSetColor, задающие толщину линии 1, убрана лишняя строка с повторной установкой желтого цвета, и убран лишний пробел в первом вызове txCircle, он теперь сильно бросается в глаза. Измененные строчки в старом коде справа выделены курсивом. Также добавлены поясняющие пустые строки.

```
void DrawValerianaVarfolomeevnaTheOrnitologist()
    {
    txSetFillColor (TX_WHITE);
    txCircle (650, 700, 30);
    txRectangle (645, 730, 655, 735);

    txSetFillColor (TX_BLACK);
    txCircle (660, 690, 3);
    txCircle (640, 690, 3);
    txCircle (650, 705, 4);
    txSetColor (TX_RED);
    txArc (640, 710, 660, 720, 180, 180);

    txSetColor (TX_WHITE);
    txSetFillColor (TX_LIGHTBLUE);
    POINT dress[4] = {{610, 735}, {690, 735},
                      {580, 870}, {720, 870}};
    txPolygon (dress, 4);

    txEllipse (630, 665, 670, 680);

    txSetFillColor (TX_YELLOW);
    txCircle ( 610, 735, 15);
    txCircle ( 690, 735, 15);
    txCircle ( 650, 782, 20);

    txSetColor (TX_YELLOW, 10);
    txLine (610, 734, 580, 760);
    txLine (690, 734, 720, 760);

    txSetColor (TX_WHITE);
    txSetFillColor (TX_YELLOW);
    txChord (620, 872, 648, 892, 180, 180);
    txChord (652, 872, 680, 892, 180, 180);
    }
```

```
 void DrawGirl3()
    {
    txSetFillColor (TX_WHITE);
    txCircle ( 650, 700, 30);
    txRectangle (645, 730, 655, 735);
    txSetFillColor (TX_BLACK);
    txCircle (660, 690, 3);
    txCircle (640, 690, 3);
    txCircle (650, 705, 4);
    txSetColor (TX_RED, 1);
    txArc (640, 710, 660, 720, 180, 180);
    txSetColor (TX_WHITE, 1);
    txSetFillColor (TX_LIGHTBLUE);
    POINT star9[4] = {{610, 735}, {690, 735}, {580,
870}, {720, 870}};
    txPolygon (star9, 4);
    txEllipse (630, 665, 670, 680);
    txSetFillColor (TX_YELLOW);
    txCircle ( 610, 735, 15);
    txCircle ( 690, 735, 15);
    txCircle ( 650, 782, 20);
    txSetColor (TX_YELLOW, 10);
    txLine (610, 734, 580, 760);
    txSetColor (TX_YELLOW, 10);
    txLine (690, 734, 720, 760);
    txSetColor (TX_WHITE, 1);
    txSetFillColor (TX_YELLOW);
    txChord (620, 872, 648, 892, 180, 180);
    txChord (652, 872, 680, 892, 180, 180);
    }
```

Эта функция тоже стала яснее, рефакторинг дал свои плоды.

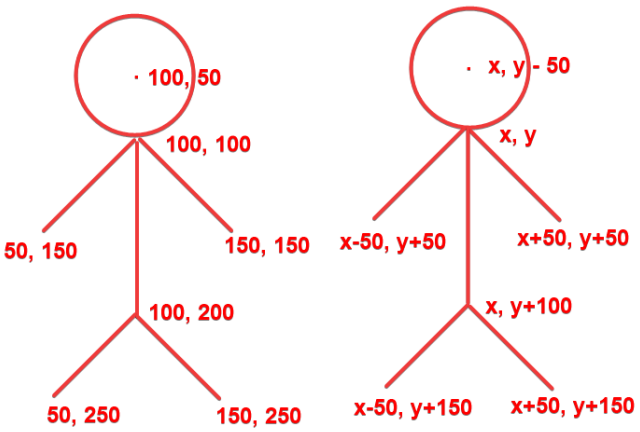
На этой стадии полезно посмотреть код ваших коллег, изучающих программирование вместе с вами, сделать ревью их кода и получить ревью от них – это называется перекрестное ревью, оно часто применяется. После этого выполните рефакторинг своего кода и порадуйтесь улучшению его качества.

1.3. Функции с параметрами

Вспомним еще одну проблему: масса трудностей с измерением координат точек объектов. Да, это было неприятно. Давайте усилим проблему масштабированием: пусть нам нужно много похожих объектов в разных местах окна, например, мы хотим нарисовать тусующихся людей. (Конечно, для этого мы напишем функцию DrawTusa.) :) Все будет просто ужасно: у человечка 7 точек и, соответственно, 14 координат, а у тусы в 10 человек этих координат будет уже 140. Полная жесть. Надо срочно что-то придумывать.

Тут можно заметить вот что. Человек в тусовке 10 и все координаты их различны, но положение рук и ног, размеры каждого человека – одинаковы. То есть, если мы возьмем и измерим координаты одного человечка в системе координат, связанной с ним же (это называется *относительной системой координат* и вводится в курсе физики, в разделе механики), то получим код, рисующий человечка в определенном заданном месте экрана.

Если вы не помните, что такое относительная система координат, то просто задайте центр человечка координатами *x* и *y*, и отсчитайте координаты его рук и ног относительно этой точки. Скажем, если правая рука человечка определялась точками (100, 100) и (150, 150), а центр человека мы решили поместить в точке (100, 100), то новые координаты будут иметь вид (x, y) и (x + 50, y + 50). Левая рука будет задана точками (x, y) и (x - 50, y + 50).



Теперь как это записать в коде? Во-первых, в заголовке функции, в скобках, ранее пустых, определим две переменные x и y. Слово int говорит о том, что это – целые числа:

```
void DrawMan (int x, int y)
    {
    ...
    }
```

Если нужны дробные числа, то вместо int нужно поставить double. Функция станет более обобщенной, чем ранее: теперь она сможет рисовать человечка в любой заданной координатами точке. Она будет работать как формула с переменными в физике или математике вместо конкретного выражения с числами. Переменные x и y будут называться *параметрами* или *аргументами* функции. Их конкретные значения будут задаваться в вызове функции, и это даст возможность при каждом конкретном вызове их менять.

Во-вторых, надо обновить прототип функции, вписав туда параметры (или аргументы). О нет, конечно же, не обновить, а провести рефакторинг прототипа! :)

В-третьих, сделать рефакторинг тела функции, заменив конкретные числовые координаты на выражения с параметрами x и y.

И, наконец, в вызове подставить координаты центра человечка в круглые скобки, ранее пустые.

Получится вот что:

```
//---------------------------------------------------------------------------
// Рисуем трех человечков тремя строчками
```

```
//---------------------------------------------------------------------------

#include "TXLib.h"

//---------------------------------------------------------------------------

void DrawMan (int x, int y);  // Прототип функции DrawMan, теперь и с параметрами!

//---------------------------------------------------------------------------

int main()
    {
    txCreateWindow (800, 600);

    DrawMan (100, 100);      // Здесь нарисуется человечек в старом месте (100, 100)
    DrawMan (300, 100);      // Здесь на новом месте (x = 300, y = 100)
    DrawMan (100, 400);      // И здесь тоже, на другом новом месте (x = 100, y = 400)
    }

//---------------------------------------------------------------------------

void DrawMan (int x, int y)  // Заголовок функции DrawMan, теперь и с параметрами,
    {                        // без регистрации и СМС!
    txSetColor (TX_WHITE);
    txSetFillColor (TX_NULL);

    txLine (x, y,       x - 50, y + 50);
    txLine (x, y,       x + 50, y + 50);
    txLine (x, y,       x,      y + 100);
    txLine (x, y + 100, x - 50, y + 150);
    txLine (x, y + 100, x + 50, y + 150);

    txCircle (x, y - 50, 50);
    }
```

Делайте все внимательно, соблюдайте синтаксис, не забывайте про запятые в нужных местах, ставьте пробелы для выравнивания выражений и соблюдайте стиль кода. Все надо делать по красоте! Иначе ни толку не будет, ни удовольствия.

Так.

Обратили внимание, что рисуется три человечка? А функция для рисования всего одна? Вот! Видите, как хорошо масштабируется? Теперь можно написать функцию DrawTusa очень легко:

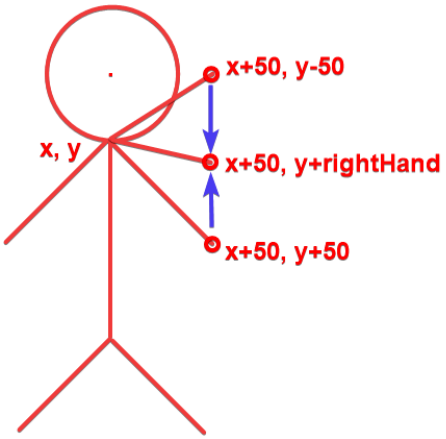
```
//---------------------------------------------------------------------------
// Аквадискотека на 10 человек!
//---------------------------------------------------------------------------

#include "TXLib.h"

//---------------------------------------------------------------------------

void DrawTusa (int x, int y); // Прототип функции DrawTusa
void DrawMan (int x, int y);  // Прототип функции DrawMan

//---------------------------------------------------------------------------

int main()
    {
    txCreateWindow (1600, 1200);

    DrawTusa (400,  400);  // Рисуем 10 человек одной строчкой
```

```
    DrawTusa (1200, 900);  // Ю-ху, еще 10
    }

//----------------------------------------------------------------------------

void DrawTusa (int x, int y)
    {
    DrawMan (x,     y);
    DrawMan (x-120, y+190);
    DrawMan (x-180, y-180);
    DrawMan (x-20,  y-140);
    DrawMan (x+130, y-90);
    DrawMan (x+60,  y+160);
    DrawMan (x+170, y-220);
    DrawMan (x-70,  y-170);
    DrawMan (x+220, y+290);
    DrawMan (x+110, y+220);
    }

//----------------------------------------------------------------------------

void DrawMan (int x, int y)                         // Заголовок функции DrawMan
    {
    txSetColor (RGB (rand(), rand(), rand()), 7);  // Это выбор случайного цвета
    txSetFillColor (TX_NULL);

    txLine (x, y,       x - 50, y + 50);
    txLine (x, y,       x + 50, y + 50);
    txLine (x, y,       x,      y + 100);
    txLine (x, y + 100, x - 50, y + 150);
    txLine (x, y + 100, x + 50, y + 150);

    txCircle (x, y - 50, 50);
    }
```

Вот! 10 человек! И где тонна строк и 140 координат, которые виделись в страшном сне? А нету их. Вместо них одна функция с параметрами, красивая и стройная, и 10 ее вызовов. Эх. Всегда бы было так легко. :)

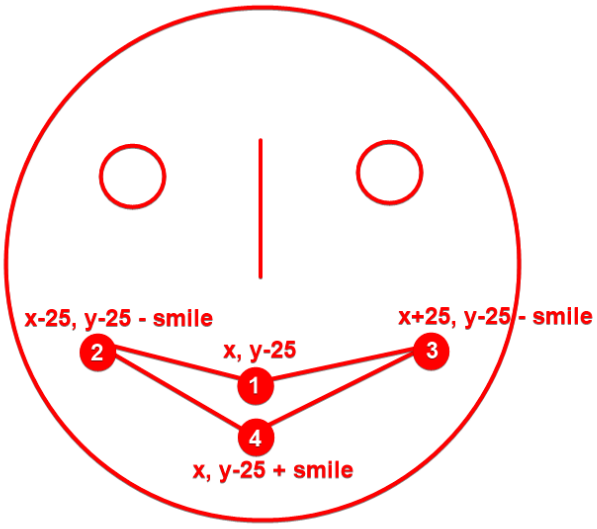
Что еще можно добавить? Цвета. В примере выше цвета задаются случайно, но цвет линий и цвет заполнения фигур можно легко передать в функцию через параметры. Для этого существует специальный тип COLORREF (на самом деле, это целое число – код цвета в формате RGB). Если объявить параметры с названиями, скажем, headColor (цвет головы), bodyColor (цвет тела человечка) и т.п., то можно раскрасить людей в тусовке в соответствии с их индивидуальностью, субкультурой и музыкальными предпочтениями. :)

Теперь попробуем изменить форму объекта – позу. На первый взгляд это сложно, но поза – это всего лишь значения относительных координат какой-то части тела. Например, если человек опустил правую руку, то координаты ее ладони будут (x + 50, y + 50). Если он вытянул ее горизонтально, то (x + 50, y + 0). Если слегка поднял, то (x + 50, y - 10). Это можно *обобщить,* введя параметр rightHand (или pravayaRuka, или thePravayaOfTheROOKA) и записав координаты ладони как (x + 50, y + rightHand). Тогда, указывая разные значения этого параметра, можно заставить человечков поднимать руки на разную высоту. Аналогично можно поступить с левой рукой. С ногами можно поступить немного по-другому, если предположить, что человек стоит всегда симметрично. Тогда, введя параметр legs, можно смещать левую ногу влево, а правую направо, задав координаты левой ноги как (x - legs, y + 150), а правой как (x + legs, y + 150). Немного лучше использовать legs/2 вместо legs (подумайте, почему).



Конечно, такой подход упрощенный, при нем меняется длина рук. Если вы работаете на студию Pixar (пока ждете приглашения на Мосфильм), то это хорошо подойдет для мамы из фильмов про супергероев. В принципе, это приемлемо и для многих мультфильмов вообще в силу их условности. Но можно подумать и о расчете координат через явно заданную длину руки, и тогда параметр rightHand будет задавать угол подъема. Да, понадобятся тригонометрические функции sin и cos (они есть в Си), и код будет сложнее, зато анатомия человечка будет сохранена. :)

Давайте попробуем изобразить на лице человечка глаза, нос и рот так, чтобы рот мог улыбаться. Лучше всего составить рот из четырех линий (см. рис). Чуть проще – из двух, если убрать точки 1 или 4 и исходящие из них отрезки. Параметр smile (улыбка) может смещать центр рта по вертикали вниз, а края рта – вверх, оставляя центр рта на прежнем месте. Попробуйте разобраться, как устроено управление ртом, из рисунка, последовательно рассматривая координаты точек в порядке 1-2-3-4 или в порядке 1-4-2-3, как будет понятнее.



Кроме того, можно задать размеры объекта по ширине и высоте, их можно назвать sizeX и sizeY или как-то по-другому (но только супер-понятно, не надо нам всяких неясных s1 и s2, пусть математики так обозначают:)). Это могут быть просто масштабные коэффициенты, умножающиеся на расстояния от центра объекта до его точек: скажем, координаты точки были (x + 50, y + 150), а стали (x + 50*sizeX, y + 150*sizeY). Для того, чтобы можно было задавать дробный масштаб, используем тип double, предназначенный для для действительных чисел.

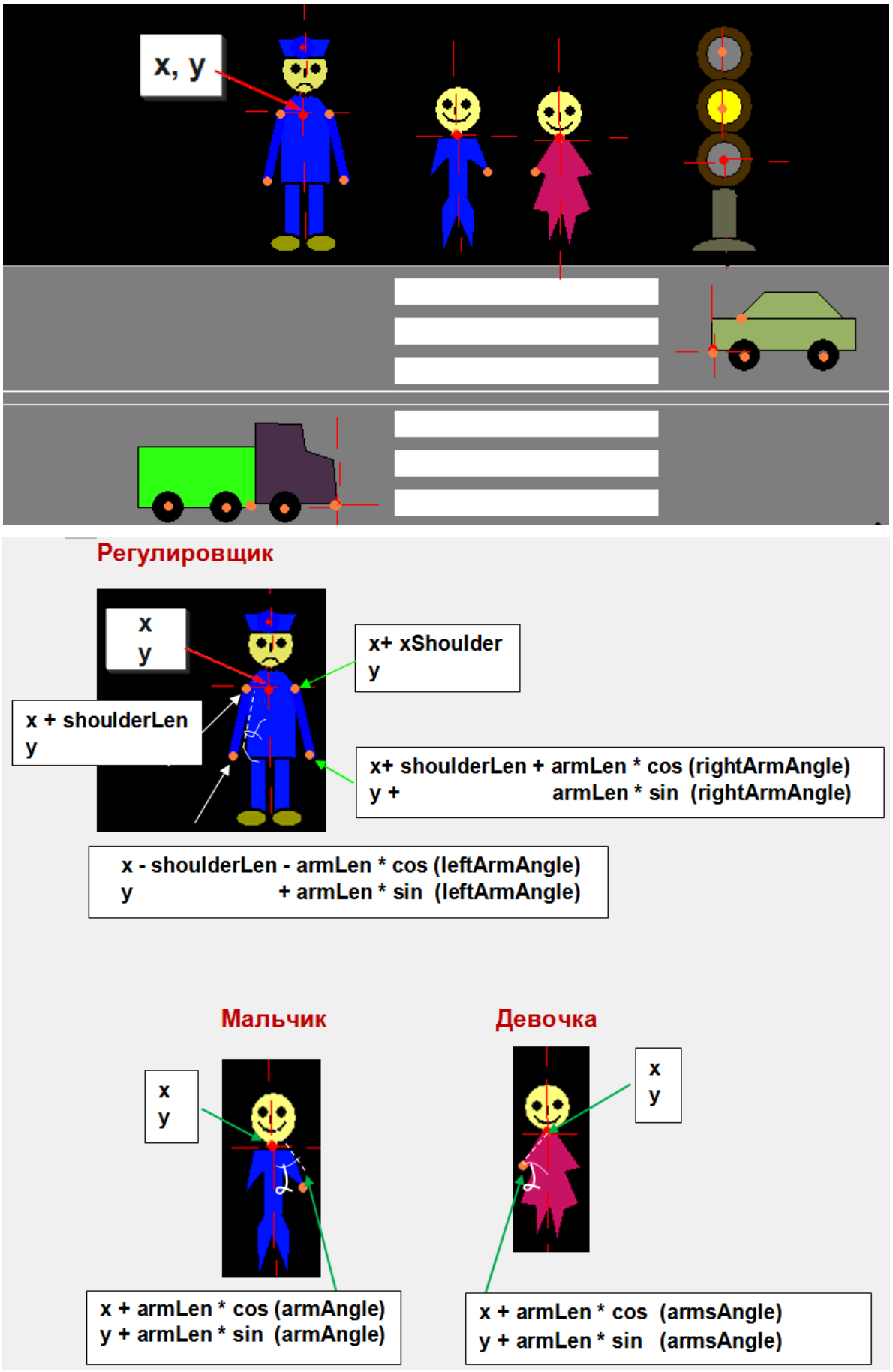
```
void DrawMan (int x, int y, double sizeX, double sizeY);
```

```
int main()
    {
    txCreateWindow (800, 700);

    DrawMan (100, 100, 1,   1);    // Стандартный человечек
    DrawMan (300, 100, 0.7, 0.7);  // Его младший брат
    DrawMan (300, 400, 0.7, 1.8);  // Старший брат
    }

void DrawMan (int x, int y, double sizeX, double sizeY)
    {
    txSetFillColor (TX_NULL);

    txLine (x, y,             x - 50*sizeX, y +  50*sizeY);
    txLine (x, y,             x + 50*sizeX, y +  50*sizeY);
    txLine (x, y,             x,            y + 100*sizeY);
    txLine (x, y + 100*sizeY, x - 50*sizeX, y + 150*sizeY);
    txLine (x, y + 100*sizeY, x + 50*sizeX, y + 150*sizeY);

    txCircle (x, y - 50*sizeY, 50*sizeY);
    }
```

Либо можно задать ширину и высоту объекта в пикселях, но тогда придется пропорционально пересчитывать координаты всех точек объекта. Подумайте, кстати, почему такой способ задания может быть несколько лучше.

Таким образом, мы можем менять взаимное расположение частей объекта, то есть его форму, позу и т.п. Что это дает? Невероятные возможности в рамках нашего рисунка. Люди в разных позах, разного вида и настроения (см. пример в функции txDrawMan библиотеки TXLib). Дома с открытыми и закрытыми окнами и дверьми, трубами разной длины и дымом над ними. Автомобили с прыгающими на неровной дороге колесами и хлопающими дверьми; птицы с разной формой крыла и самолеты с убирающимися шасси; котики с разным расположением лап, хвостов и усов; яблони с падающими яблоками, ждущие новых Ньютонов, и Ньютоны с параметрами положения глаз для слежения за яблоками; елки с глазами, удивленно распахивающимися, когда их приходит рубить удивленный лесоруб; лесорубы, с помощью пары параметров быстро убегающие от таких стремных елок. Да что угодно. Параметры – это свобода. И еще параметры – это будущее движение: мы будем их активно использовать при движении персонажей в сценах мультфильма.

Вот еще одна схема расчета персонажей, основанная на углах подъема (отклонения) рук и ног персонажей. Она более анатомична, так как при подъеме или опускании рук не меняется их длина. Но для этого пришлось применить тригонометрические функции. Кстати, автор этих персонажей решил в каждой функции рисовать поверх самих фигур героев еще и красный кружок точно в точке (x, y) для удобства отладки. Если рисуемый персонаж оказался не там, где надо, то прежде всего можно проверить положение этого красного кружочка. Если он не там, значит, при вызове функции параметры были заданы неверно. Если он там, а герой не там, значит, значения при вызове переданы верно, а вот в самой функции что-то не так с расчетами координат частей персонажа. Когда объект в порядке, этот красный кружочек можно удалить из кода.



Сколько параметров должно быть у функции? Это неоднозначный вопрос. Рассмотрим такое вот сравнение. Параметры "настраивают" функцию на нужный режим работы, так же, как, скажем, пульт от телевизора настраивает его на нужный канал, громкость, яркость и прочее. Сколько кнопок должно быть у пульта? Если одна – можно выбирать только канал, и все; этого мало. Но

зато пульт простой, легко пользоваться. Если очень много кнопок – доступна очень гибкая настройка, но пользоваться сложнее. Поэтому четкого ответа дать нельзя, нужен баланс, и он еще зависит от того, кто будет пользоваться функцией. Для облегчения этого в C++ есть такая вещь, как параметры по умолчанию – в прототипе (и только в нем!) функции у "хвоста" параметров можно задать значения, которые будут использованы, если в вызове эти значения отсутствуют. Таким образом, функцию с десятью параметрами можно вызвать, указав, например, всего два из них. Пропустить при вызове несколько параметров, а затем указать, например, самый последний, не получится, поэтому нужно располагать параметры по степени их важности. Вот пример прототипа функции, он основан на функции txDrawMan из библиотеки TXLib.

```
void DrawManEx (int x, int y,                                    // Координаты, без значений по умолчанию
                double sizeX = 1, double sizeY = 1,              // Размеры, если не указаны, то масштаб = 1
                COLORREF color = TX_WHITE,                       // Цвет, по умолчанию - белый
                double handL = 0, double handR = 0, double twist = 0, // Контроль позы
                double head = 0, double eyes = 0.8, double wink = 0,  // Голова и глаза
                double crazy = 0, double smile = 1.0,            // Эмоции
                double hair = 0, double wind = 0);               // Прическа
```

Еще раз: в заголовке функции, в ее определении, указывать значения по умолчанию не надо – будет ошибка. Указывайте только в прототипе.

Полный вызов, без использования значений параметров по умолчанию, выглядит так:

```
DrawManEx (125, 250, 200, 200, TX_WHITE, 0, 0, 0, 0, 0.8, 0, 0, 1.0, 0, 0);
```

Но наличие значений по умолчанию в прототипе дает возможность вызывать эту функцию так:

```
DrawManEx (125, 250, 200, 200, TX_WHITE, 0, 0, 0, 0, 0.8, 0, 0, 1.0);
```

И так:

```
DrawManEx (125, 250, 200, 200, TX_WHITE);
```

И даже так:

```
DrawManEx (125, 250);
```

Также для функции со сложным вызовом можно сделать парную ей функцию-помощник (helper), принимающую меньше параметров, и сразу вызывающую более сложную функцию. Таких помощников может быть несколько. При наличии помощников обычно к более сложной функции дописывается суффикс Ex (от "Extended"), а функция-помощник называется так, как раньше называлась исходная функция. Ну или к помощнику дописывается слово "Helper", или вставляется слово "Simple" и т.п., так, чтобы увеличить понятность текста.

```
void DrawManSimple (int x, int y, COLORREF color);

int main()
    {
    // ...

    DrawManSimple (125, 250, TX_WHITE);   // Простой способ рисования человечков, годится для
    DrawManSimple (325, 250, TX_YELLOW);  // "массовки", а для главных героев - DrawManEx
    }
```

```
void DrawManSimple (int x, int y, COLORREF color)
    {
    DrawManEx (x, y, 200, 200, color, 0, 0, 0, 0, 0.8, 0, 0, 1.0, 0, 0);
    }
```

[[Врезка: зачем вообще увеличивать понятность? Литературное программирование Д.Кнута]]

1.3.1. Задание

Для каждой функции, написанной вами, добавьте параметры положения на экране (x, y), цвета, размера, а также не менее трех "индивидуальных" параметров, доступных только в данной функции (а лучше побольше). Вызовите эти функции много раз, чтобы показать новые возможности ваших функций, указывая значения параметров в виде чисел при вызове. Например, если у дома, нарисованного вами, можно менять степень открытия двери, и высоту печной трубы, то нарисуйте деревеньку из нескольких разноцветных домов разного размера с разными дверьми и трубами (конечно, сделайте для этого отдельную функцию). Если вы нарисовали елку с глазами, то составьте из елок с глазами лес, где одни елки будут спать, другие удивляться, третьи коситься на прохожих, четвертые будут просто и прямо смотреть на незнакомца: зачем пришел, добрый человек? По лесу прогуляться? А топор зачем прячешь??? А НУ БРОСЬ, СКАЗАЛИ!!! (Это может быть полезно для экологического мультфильма, да.) Про функции типа DrawTusa с рисованием разных человечков мы уже говорили раньше. В этот момент экран может быть перегружен рисунками, это не страшно. Пусть ваш "внутренний художник" порадуется. :)

Как вы поняли, у нас опять рефакторинг проекта – введение параметров в функции. Как часто программисты добавляют в функции новые параметры? Как говорят, приблизительно 300к раз в наносекунду. :) В общем, часто. :)

1.3.2. Пример, который... (ну вы поняли)

...вы поняли.

Этот пример входит в состав примеров TXLib (Пример: Функции с параметрами).

```
#include "TXLib.h"

//------------------------------------------------------------------------

void DrawMan    (int x, int y, int sizeX, int sizeY, COLORREF color,
                 double hand = 0, double legs = 0, double head = 0, double twist = 0);
void DrawEarth  (int x, int y, int sizeX, int sizeY, COLORREF color);
void DrawFlag   (int x, int y, int sizeX, int sizeY, COLORREF color, COLORREF bkColor);
void DrawHello  (int x, int y, const char* text, int size, COLORREF color);
void DrawFrame  (int sizeX, int sizeY, int size, COLORREF color);

//------------------------------------------------------------------------

int main()
    {
    txCreateWindow (800, 600);

    DrawFrame (800, 600, 10, TX_WHITE);
    DrawHello (400, 480, "Hello, world!", 60, TX_LIGHTGREEN);
```

```
    DrawEarth (400, 300, 400, 300, TX_LIGHTCYAN);

    DrawFlag  (400, 150, 50, 75, TX_YELLOW, TX_TRANSPARENT);
    DrawMan   (385, 150, 20, 40, TX_YELLOW, 0, 0, 0);

    txTextCursor (false);
    return 0;
    }

//---------------------------------------------------------------------------

void DrawMan   (int x, int y, int sizeX, int sizeY, COLORREF color,
                double hand, double legs, double head, double twist)
    {
    txSetColor     (color);
    txSetFillColor (color);

    txLine (x, y - (0.35 + twist) * sizeY, x, y - 0.7*sizeY);

    txLine (x, y - (0.35 + twist) * sizeY, x - (0.5 + legs) * sizeX, y);
    txLine (x, y - (0.35 + twist) * sizeY, x + (0.5 + legs) * sizeX, y);

    txLine (x, y - 0.65*sizeY, x - sizeX/2,   y - 0.4*sizeY);
    txLine (x, y - 0.65*sizeY, x + sizeX/1.2, y - (0.7 + hand) * sizeY);

    txCircle (x, y - sizeY + (0.3 + head) * sizeX, 0.3*sizeX);
    }

//---------------------------------------------------------------------------

void DrawEarth (int x, int y, int sizeX, int sizeY, COLORREF color)
    {
    txSetColor (color);

    int r = sizeX/2;
    while (r >= 0)
        {
        txEllipse (x - r, y - sizeY/2, x + r, y + sizeY/2);
        r -= sizeX/9;
        }

    r = sizeY/2;
    while (r >= 0)
        {
        txEllipse (x - sizeX/2, y - r, x + sizeX/2, y + r);
        r -= sizeY/6;
        }

    txLine (x - sizeX/2, y, x + sizeX/2, y);
    }

//---------------------------------------------------------------------------

void DrawFlag (int x, int y, int sizeX, int sizeY, COLORREF color, COLORREF bkColor)
    {
```

```
    txSetColor     (color);
    txSetFillColor (bkColor);

    txLine      (x, y, x, y - sizeY);
    txRectangle (x, y - sizeY/2, x + sizeX, y - sizeY);

    txSelectFont ("Times New Roman", 20);
    txTextOut (x + sizeX/2, y - sizeY*7/8, "C++");
    }

//---------------------------------------------------------------------------

void DrawHello (int x, int y, const char* text, int size, COLORREF color)
    {
    txSetColor (color);

    txSelectFont ("Times New Roman", size);
    txSetTextAlign (TA_CENTER);

    txTextOut (x, y, text);
    }

//---------------------------------------------------------------------------

void DrawFrame (int sizeX, int sizeY, int size, COLORREF color)
    {
    txSetColor (color);
    txSetFillColor (TX_TRANSPARENT);

    txRectangle (size, size, sizeX - size, sizeY - size);
    }
```

1.3.3. Ревью кода

1.3.3.1. Первый пример

Функции с параметрами дело непростое, поэтому давайте рассмотрим несколько примеров ревью кода. Первый:

	Фрагмент
`int main()`	1
`    {`	1
`    ...`	1
`    int x = 0;`	2
`    int y = 525;`	2
`    int n = 60;  // размер кроны`	2
`    double s = 1;`	2
`    COLORREF trunk = RGB (185, 122, 87), crown = COLORREF (TX_GREEN);`	2
`    DrawTree (x, y, n, s, trunk, crown);`	2
`    double AngLA, AngRA, AngLL, AngRL;`	3
`    AngLA = 150;`	3

```
    AngRA = 160;                                                                    3
    AngLA = 140;                                                                    3
    AngRL = 130;                                                                    3
    COLORREF robot = RGB (200, 191, 223);                                           3
    COLORREF eyes  = COLORREF (TX_BLACK);                                           3
    COLORREF smile = COLORREF (TX_BLACK);                                           3
    x = 250;                                                                        3
    y = 370;                                                                        3
    DrawRobot (x, y, s, AngLA, AngRA, AngLL, AngRL, robot, eyes, smile);            3

    // (далее все в таком же духе)
    }
```

Здесь в функции main вызываются функции DrawTree и DrawRobot, определенные в коде где-то ниже. Сразу ли ясен ли код с первого взгляда?

Код – такой-то такой... с душком. Как говорит Мартин Фаулер, this code smells.

Во-первых, здесь беда с названиями переменных. С иксом и игреком еще более-менее понятно, но что такое s, AngLA, AngRA, AngLL и AngRL? Неясно. Если залезть в функции DrawTree и DrawRobot, разобраться, что значат эти параметры, то смысл прояснится, но такое залезание придется делать каждый раз, когда вы отвлечетесь от текущего фрагмента кода – то есть в больших программах достаточно часто. Через месяц-другой вы вообще не будете помнить смысл этих имен (хотите проверить? готов поспорить с вами), и влезание в старый код будет сопровождаться характерным чертыханием на самого себя, сдавленными криками “какой дурак так это назвал” и прочими признаками самокритики. У программистов есть ироничное выражение “reverse engineering по исходному коду”, это когда текст есть, но разбираться приходится так, как будто перед нами машинный код в двоичной форме.

(...Вот как вы думаете: что означает переменная z из одного из исходников, на который я сейчас смотрю? И чем она отличается от z1? Ох. Это учебный код, не рабочий. Нет, это не z-координаты. Это размер объекта, вы не поверите, по X и Y. Нет, это не я его писал. Это студент писал. В школе его учили плохо, в вузе (не МФТИ) тоже. Теперь он не видит отличий в именах, сам путается. Ох. Книжку МакКоннелла ему читать сложно. Ну, может, хоть эту прочтет.)

Некоторые короткие имена переменных похожи на цифры: l (L малое) на 1 (единицу), O (O большое) на 0 (ноль). Современные редакторы кода по-разному подсвечивают цифры и буквы, но все равно такие имена не надо использовать. Если l используется для длины, то лучше length или len. Вообще трижды подумайте, прежде чем использовать однобуквенные имена. Знаете, например, как называется величина скорости в известном игровом движке Unreal Engine? Velocity, а не просто “v”. Встречается оно там много раз, и людям не лень писать его каждый раз. Кстати, название скорости в физике взято именно от этого слова. Но физики и математики пишут на доске и бумаге, им позволительно сильно сокращать слова. Нам – нет.

Далее. Переменная n во фрагменте 2 помечена комментарием, что это “размер кроны”, и, видимо, автор кода считает, что долг перед смыслом он выполнил. Но какая связь именно имени с этим смыслом? Видимо, она в том, что в слове “крона” (crown) есть буква n? Тем более что в грамотном коде имя “n” если для чего и используется, то для количество чего-либо, от слова “number”, и только как составная часть названия, например nTrees – количество деревьев (number of trees), nElems – количество элементов, nStrings – количество строк и т.д. Аргумент, что здесь стоит пояснение в виде комментария ("размер кроны"), несостоятелен.

Комментарий может устареть или потеряться, и это никто не заметит, потому что за комментариями не следит компилятор; у него может сбиться кодировка, он будет отображен “крокозяблами” и его не прочтут; его не поймет нерусскоязычный член команды; да и вообще, зачем нагружать память программистов в двойном размере – запоминать отдельно имена переменных и отдельно их смысл? Это даже противнее, чем зубрить что-то наизусть. Программирование – для творческой работы, а не для зубрежки. Поэтому не надо создавать себе и коллегам необходимость запоминать смысл переменных или, чертыхаясь, лазить по комментариям. Надо просто назвать переменные нормально. Например, вот слегка отрефакторенный код без всяких пояснений:

```
                                                                                    Фрагмент

const COLORREF TrunkColor = RGB (185, 122, 87),                                     1
               CrownColor = COLORREF (TX_GREEN),                                    1
               RobotColor = RGB (200, 191, 223),                                    1
               EyesColor  = COLORREF (TX_BLACK),                                    1
               SmileColor = COLORREF (TX_BLACK);                                    1
int main()
    {
    ...

    int x         = 0,                                                              2
        y         = 255;                                                            2
    int crownSize = 60;                                                             2
    double size   = 1;                                                              2
                                                                                    2
    DrawTree  (x, y, size, crownSize, TrunkColor, CrownColor);                      2

    double angleLeftArm  = 150,                                                     3
           angleRightArm = 160,                                                     3
           angleLeftLeg  = 140,                                                     3
           angleRightLeg = 130;                                                     3
    x = 250;                                                                        3
    y = 370;                                                                        3
                                                                                    3
    DrawRobot (x, y, size, angleLeftArm, angleRightArm, angleLeftLeg,               3
angleRightLeg,                                                                      3
               RobotColor, EyesColor, SmileColor);
    ...
    }
```

И с именами все ясно.

Переменные, отвечающие за цвета (trunk, robot, eyes и smile) сделаны константами, вынесены из main в начало файла, и названы с больших букв, чтобы подчеркнуть, что они могут использоваться везде в коде (см. фрагмент 1). У них уточнены названия, добавлено слово Color. Их названия теперь говорят сами за себя.

Переменные внутри функции main обозначены ясными названиями (фрагменты 2 и 3):

Переменная	**Имя переменной, если разбить на части**	**Машинный перевод Google**	**Машинный перевод Yandex**
crownSize	crown Size	размер короны	размер коронки

Переменная	**Имя переменной, если разбить на части**	**Машинный перевод Google**	**Машинный перевод Yandex**
size	size	размер	размер
angleLeftArm	angle Left Arm	угол левой руки	наклоните левую руку
angleRightArm	angle Right Arm	угол правой руки	наклоните правую руку
angleLeftLeg	angle Left Leg	угол левой ноги	наклоните левую ногу
angleRightLeg	angle Right Leg	угол правой ноги	наклоните правую ногу

Здесь специально взят формальный машинный перевод, как будто мы совсем не знаем английского. И он дает приемлемые варианты, хотя крона перевелась то как корона, то как коронка. А вот если назвать переменную treeCroneSize, то перевод будет уже точнее.

Далее. Всем переменным были сразу присвоены начальные значения, чтобы избежать логических ошибок (см. начало фрагмента 3). В исходном примере с переменными AngLA, AngRA, AngLL, AngRLтак не было сделано. Сначала переменные объявлялись без начальных значений, а потом им эти значения присваивались. Когда переменная объявляется без начального значения, то ее значение становится неопределенным, тем, которое осталось в памяти от предыдущей работы других программ и алгоритмов, то есть практически случайным. Это привносит в работу программы хаос, который отследить очень трудно. Кстати, в нашем примере такой хаос был. Посмотрите внимательнее на исходный пример, переменной AngLL так и не было присвоено никакого значения. Если кажется, что в этом случае значение будет ноль, то нет, это не так. Значение в этом случае останется неопределенным. (Погуглите “Задача про Буратино и яблоки для программистов”, она вот про это.)

Задание начальных значений переменных, по-научному инициализация – мощное средство гарантии надежности программ. В некоторых языках даже запрещено создавать переменные без инициализации. Язык Си в этом либерален, но мы, программисты-профессионалы, себе такой вредной свободы не позволяем. Пусть другие неделями ищут ошибки от отсутствия инициализации, а у нас они даже возникать не будут.

Кроме того, в ходе рефакторинга приведена в общий порядок последовательность параметров в вызовах функций DrawTree и DrawRobot:

1. сначала координаты x и y, которые есть у всех рисовательных функций,
2. потом размер size, который есть у многих,
3. потом параметры, которые у каждой функции будут свои собственные.

Разумеется, определения и прототипы этих функций тоже были соответствующим образом изменены.

Все имена также приведены в порядок: то, что определяется внутри функций – с маленькой буквы (x, y, size, angleLeftArm и т.п.), а то, что определяется сверху, для всего файла в целом – с большой (TrunkColor, CrownColor и т.п.).

Что ж, код стал яснее и четче. Но осталась одна проблема – не очень понятно, зачем автору кода захотелось создавать переменные для всего лишь одного вызова функции. В примере выше все переменные, кроме size, используются только для тех вызовов функций DrawTree или DrawRobot, которые идут сразу после них. Кажется, что x и y не таковы, но, если

присмотреться, их значение заново задается перед каждым вызовом очередной функции. То есть они тоже "одноразовые".

Так обычно приходится поступать, если значения переменных, например, x и y, используются в нескольких вызовах функций, при этом они одинаковы или связаны логически друг с другом. Например, если было бы так:

```
    int x = 100,
    int y = 525;
    ...
    DrawTree (x + 100, y - 200, size, crownSize, TrunkColor, CrownColor);
    DrawTree (x, y + 100, size, crownSize, TrunkColor, CrownColor);
    DrawTree (x - 70,  y, size, crownSize, TrunkColor, CrownColor);
    DrawTree (x + 150, y + 150, size, crownSize, TrunkColor, CrownColor);
```

Или, если чуть изменить форматирование, отойдя от формальных правил (один пробел после запятой), и выровнять код по вертикали – то логическая связь станет яснее, а код – лучше:

```
    int x = 100;
    int y = 525;
    ...
    DrawTree (x + 100, y - 200, size, crownSize, TrunkColor, CrownColor);
    DrawTree (x,       y + 100, size, crownSize, TrunkColor, CrownColor);
    DrawTree (x - 70,  y,       size, crownSize, TrunkColor, CrownColor);
    DrawTree (x + 150, y + 150, size, crownSize, TrunkColor, CrownColor);
```

Тут иксы и игреки, выстраивающиеся в столбцы, буквально кричат о логической взаимосвязи (посмотрите также пример использования к функции txDrawMan, там вообще в виде таблички сделано). И хорошо, что кричат о взаимосвязи, это позволяет быстрее понять логику кода. Она такова, что несколько деревьев растут вокруг единого центра с координатами x и y. Изменив значения этих переменных, мы быстро переместим группу деревьев в другое место. Полезная вещь.

Но! Лучше было бы сделать так – создать новую функцию DrawManyTrees с параметрами x и y, и вызвать ее одной строкой, сняв все вопросы по этому коду:

```
int main()
    {
    ...
    DrawManyTrees (100, 525, 60);
    ...
    }

void DrawManyTrees (int x, int y, double size, int crownSize, COLORREF TrunkColor, COLORREF
CrownColor)
    {
    DrawTree (x + 100, y - 200, size, crownSize, TrunkColor, CrownColor;
    DrawTree (x,       y + 100, size, crownSize, TrunkColor, CrownColor);
    DrawTree (x - 70,  y,       size, crownSize, TrunkColor, CrownColor);
    DrawTree (x + 150, y + 150, size, crownSize, TrunkColor, CrownColor);
    }
```

И все дела. Подобный подход мы применяли выше в функции DrawTusa, она тоже была удобная и ясная. Ясный код – нет хлопот.

Если подставить значения "одноразовых" переменных в вызовы функций, то получится так:

```
const COLORREF TrunkColor = RGB (185, 122, 87),
               CrownColor = COLORREF (TX_GREEN),
               RobotColor = RGB (200, 191, 223),
              EyesColor  = COLORREF (TX_BLACK),
              SmileColor = COLORREF (TX_BLACK);
int main()
    {
    ...

    double size= 1;

    DrawTree  (0,   255, size, 60,                 TrunkColor, CrownColor);
    DrawRobot (250, 370, size, 150, 160, 140, 130, RobotColor, EyesColor, SmileColor);
    ...
    }
```

И все.

Однако, тут есть такой момент. Если вы путаетесь со смыслом параметров в вызовах функций (что означает число 150?), то, в обучающих целях, вы можете делать такие вспомогательные "одноразовые" переменные. Но обязательно инициализируйте их. В дальнейшем доля таких поясняющих переменных у вас сократится.

Вот еще один пример кода с душком:

```
void DrawHouse (int x, int y, int h, int d, COLORREF color_house, COLORREF color_window)
    {
    txSetColor (color_house);
    txSetFillColor (color_house);
    txRectangle (x - d / 2, y + h, x + d / 2, y + h + d);
    txSetColor (color_window, 4);

    int p =  y + h + 20;
    while (p <=  y + h + d - 20)
        {
        txLine (x - d / 2, p, x + d / 2, p);
        p = p + 20;
        }

    txSetColor (color_house, 3);
    txSetFillColor (color_window);
    txRectangle (x - d * 0.3, y + h + d / 5, x - 2,       y + h + d * 0.8);
    txRectangle (x + 2,       y + h + d / 5, x + d * 0.3, y + h + d * 0.8);

    txLine (x,  y, x - d, y + h);
    txLine (x,  y, x + d, y + h);
    txLine (x - d, y + h, x + d, y + h);
    txFloodFill (x, y + h / 2);
    }
```

Хоть тут все аккуратно выровнено, хорошо расставлены пробелы, пустые строки и отступы, совершенно неясен смысл параметров h и d и переменной p. Что это, высота, диаметр и...

давление?.. А диаметр чего? В доме круглых элементов нет... Загадка. Запустив программу, видим:

Что ж, красивый дом. У нас есть выбор – функция несложная, можно по рисунку взять и с нуля написать код с нормальными именами. Или разобраться, что означают переменные, но только тогда надо сразу их правильно переименовать. Иначе понимание ускользнет спустя время, и придется опять решать ребусы.

1.3.3.2. Второй пример

```
int main()
    {
    ...
    double ang = 120 * 3.141592 / 180;

    DrawHouse (540, 320, 1, ang, 650, 400, 560, 390);
    DrawHouse (640, 420, 1, ang, 750, 500, 660, 490);
    DrawHouse (440, 220, 1, ang, 550, 300, 460, 290);
    ...
    }

void DrawHouse (double x, double y, double size, double ang, double x_d, double y_d,
                double x_w, double y_w) // коорд. левого верхнего угла дома, коэфф. размера дома
    {
    DrawBase (x, y, size, ang);
    DrawDoor (x_d, y_d, size);
    DrawWindow (x_w, y_w, size);
    }

void DrawBase (double x, double y, double size, double ang)
    {
    ...
    txRectangle (x, y, x + 250 * size, y + 220 * size);
    ang = 3.141592 / 2 - ang / 2;
    txSetColor (door);
    POINT roof[3] = {{x, y}, {x + 250 * size/2, y - tan (ang) * 250 * size/2}, {x + 250 * size,
y}};
    txPolygon (roof, 3);
    ...
    }
```

```
void DrawDoor (double x_d, double y_d, double size)
    {
    ...
    txRectangle (x_d, y_d, x_d + 70 * size, y_d + 130 * size);
    }

void DrawWindow (double x_w, double y_w, double size)
    {
    ...
    txRectangle (x_w, y_w, x_w + 70 * size, y_w + 110 * size);
    ...
    txLine (x_w + 35 * size, y_w, x_w + 35 * size, y_w + 110 * size);
    txLine (x_w, y_w + 55 * size, x_w + 70 * size, y_w + 55 * size);
    }
```

Посмотрим на параметры функции DrawHouse. Их имена не очень ясны, но – супер! – унас же есть поясняющие комментарии! Что они говорят? "Коорд. левого верхнего угла дома, коэфф. размера дома". Так, позвольте. Это объяснение параметров x, y и size (размер), тут и так все ясно из названий, говорит капитан Очевидность. А как же таинственные ang, x_d, y_d, x_w и y_w? А комментарии о них ничего не говорят. Автор забыл вписать пояснения, а компилятор – помним – не проверяет комментарии. Вот и ловушка: комментарии вроде бы есть, а толку от них нет. Именно поэтому в современном коде комментарии используются очень редко, только тогда, когда никаким внятным способом нельзя выразить мысль через конструкции языка программирования. Если можно сделать функцию с ясным названием, переменную с ясным названием, константу с ясным названием – конечно же, нужно сделать именно так.

Кроме того, пользоваться такой функцией неудобно. При вызове приходится указывать отдельно координаты базовой части дома (x и y), двери (x_d и y_d – так вот что значат эти параметры!) и окна (x_w и y_w). Шесть чисел. Да еще и таинственный ang (angle?), участвующий в рисовании крыши, видимо, это угол ее наклона. Чтобы переместить дом или нарисовать другой такой же, надо изменить все эти шесть чисел, иначе он развалится. В коде выше в функции main это делается два раза мучительным пересчетом всех шести координат. Лучше бы было сделать параметрами функции координаты одной базовой точки дома, задающей его центр или середину его основания, а координаты остальных его частей сделать относительными, отсчитывающимися от этой базовой точки. Тогда, перемещая дом, мы изменяли бы только два числа, и части дома никогда не отрывались бы друг от друга:

```
int main()
    {
    ...
    double ang = 120 * 3.141592 / 180;

    DrawHouse (540, 320, 1, ang);
    DrawHouse (640, 420, 1, ang);  // Изменили всего два числе – переместили
    DrawHouse (440, 220, 1, ang);  // Изменили всего два числе – переместили
    ...
    }

void DrawHouse (double x, double y, double size, double ang)
    {
    DrawBase   (x, y, size, ang);  // Таинственный ang оставим на рефакторинг автору кода :)
```

```
    DrawDoor   (x + 110*size, y + 80*size, size);  // Расчет относительно базовой точки дома
    DrawWindow (x +  20*size, y + 70*size, size);  // Окно не отвалится, дверь не оторвется
    }
```

Дополнительно, здесь фигурирует "магическое число" 3.141592, в котором, несомненно, все сразу должны узнать константу Пи. Ну, предположим, все вокруг помнят все цифры числа Пи, это, наверное, несложно. Но не все числа так легко узнаваемы с первого взгляда. Например, число 2.718281828 (2.7 + два года рождения Льва Толстого. О какая межпредметная связь!). Это число e, основание натуральных логарифмов. Ну хорошо, это тоже помним. А как насчет $6.67430*10^{-11}$, $1.380649*10^{-23}$ и $8.98755 *10^{9}$? А меж тем это гравитационная постоянная, постоянная Больцмана и постоянная Кулона. Стыдно не знать такие вещи, господа будущие студенты Физтеха. :) Но на самом деле стыдно и неграмотно каждый раз напрямую писать такие числа в коде. Это делает код практически нечитаемым и дополнительно – все начинают побаиваться этих чисел, изменишь – вдруг что сломается. А изменять иногда надо, например, при переходе в другую систему счисления. Поэтому напрямую конкретные числа не используют вообще. Вместо этого делают константы с понятными названиями, причем не только для таких некруглых чисел, но и вообще для всех величин, регулирующих важные вещи в программе:

```
#include "TXLib.h"

const double   Pi             = 3.141592;     // Но вообще-то уже есть стандартная константа M_PI и txPI
const double   E              = 2.718281828;  // А также есть M_E (#include "math.h" для M_PI и M_E)
const double   GravityCoeff   = 6.67430e-11;
const double   BoltzmannCoeff = 1.380649e-23;
const double   CoulombCoeff   = 8.98755e9;    // Кулон - француз, поэтому такое вот с именем ¯\_(ツ)_/¯

const int      NHouses        = 10;           // N = "Number of". Частое сокращение
const int      NumTrees       = 20;           // Или такое

const COLORREF HouseColor     = RGB (185, 122, 87);
const COLORREF SkyColor       = TX_BLUE;      // Вдруг придется менять цвет на TX_LIGHTBLUE, например

...

int main()
    {
    txCreateWindow (500, 500);
    ...
    }

...
```

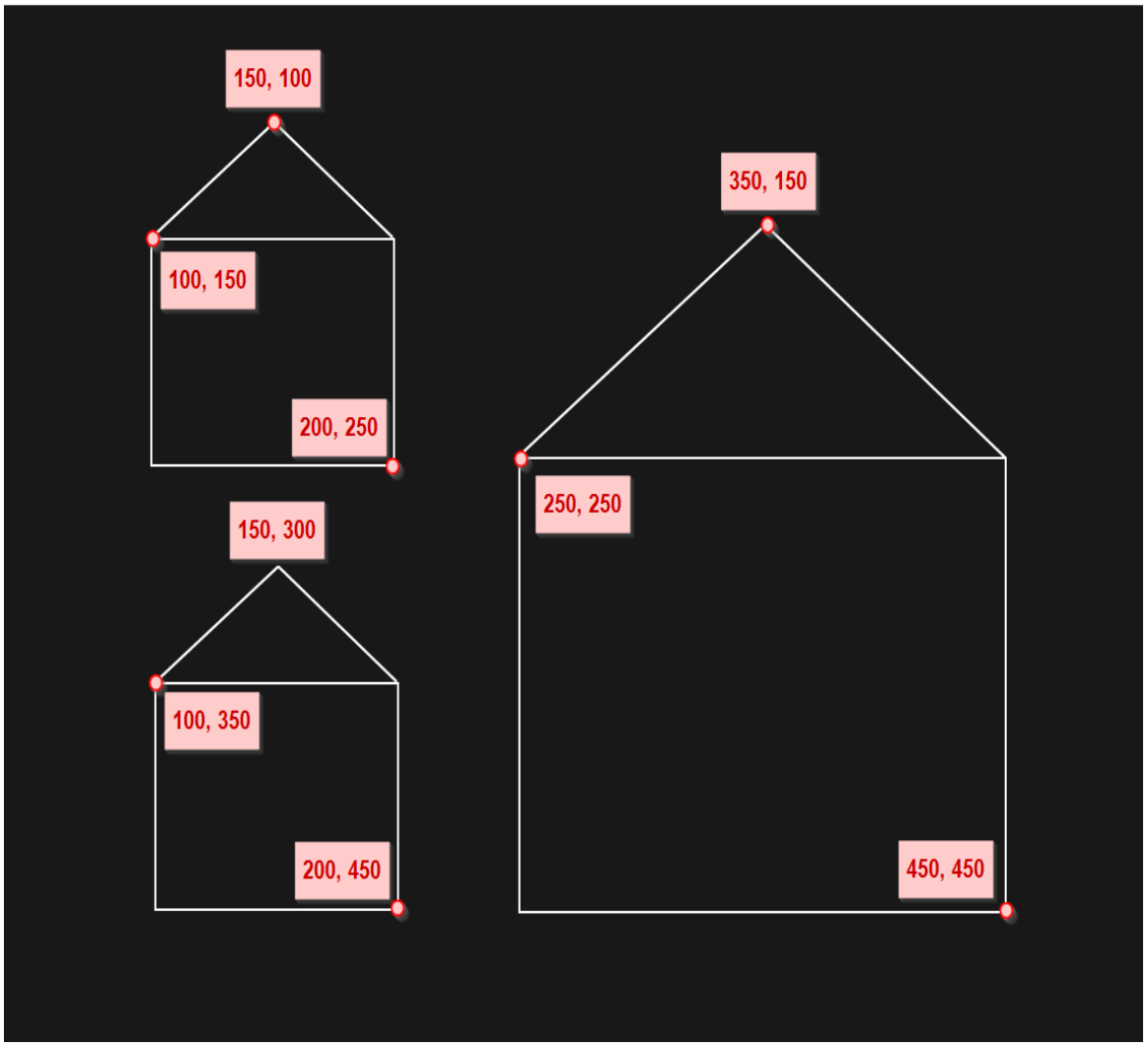
Это делает код ясным и легко регулируемым: надо глобально изменить цвет неба, дома или другую важную величину в программе – добро пожаловать на самый верх файла, в раздел констант, и меняй себе на здоровье. Только не измени случайно Вселенную, изменив число Пи.

1.3.3.3. Третий пример

Этот пример иллюстрирует очень распространенную ошибку разработки программ. Настолько распространенную, что автор долгое время давал это задание в виде контрольной работы по

теме "Функции с параметрами". И настолько важную, что на ее основе сформулировали один из золотых принципов программирования "DRY". О нем ниже.

Задание такое: дан черно-белый рисунок, напишите полную программу, рисующую его оптимальным образом (формально – наименьшим количеством операторов), используя только функции txCreateWindow и txLine:



Вообще, к понятию "наименьшее количество операторов" надо подходить осторожно, часто такие "наименьшие" программы очень трудно читать из-за переусложненного кода, и в общем случае для профессионального программирования такой подход неприемлем. Но для данного конкретного задания это определение оптимальности не мешает.

Ох. Что только не пишут в ответ на это задание.

Вот, скажем так, слишком... эээ, как бы это сказать... прямолинейное решение:

```
#include "TXLib.h"

int main()
    {
    txCreateWindow (500, 500);

    txLine (100, 150, 150, 100);
    txLine (200, 150, 150, 100);
    txLine (100, 150, 200, 150);
```

```
    txLine (200, 150, 200, 250);
    txLine (200, 250, 100, 250);
    txLine (100, 250, 100, 150);

    txLine (100, 350, 150, 300);
    txLine (200, 350, 150, 300);
    txLine (100, 350, 200, 350);
    txLine (200, 350, 200, 450);
    txLine (200, 450, 100, 450);
    txLine (100, 450, 100, 350);

    txLine (250, 250, 350, 150);
    txLine (450, 250, 350, 150);
    txLine (250, 250, 450, 250);
    txLine (450, 250, 450, 450);
    txLine (450, 450, 250, 450);
    txLine (250, 450, 250, 250);

    return 0;
    }
```

Его автору было не лень измерить координаты пятнадцати точек, написать восемнадцать вызовов функции txLine, сделав работу на уровне не выше первого занятия по TXLib. А что, программа работает, код в порядке, чего еще надо?

(И это если мы возьмем код с нормальным форматированием, а было и... скажем так, всякое.)

Давайте просто попробуем помасштабировать наше решение. Простейшее действие для анализа кода, которое мы уже не раз применяли.

Три домика – 18 строк.

Шесть домиков – 36 строк.

Десять домиков – о, нет! 180 строк! Задание невыполнимо для контрольной! (Тут, кстати, автор, как преподаватель, согласен. Если правильное решение и вправду должно быть таким длинным, то для контрольной в начале обучения это и впрямь много.)

Все потому что проще сделать, а потом подумать, чем подумать, а потом сделать. (Или не думать вообще.)

Ок, вижу, что ученики, сопя, выписывают кучи вызовов txLine. Даю подсказку: "вспомните, что контрольная – на функции с параметрами!".

А, так? Он функции хочет? Ладно, сделаем ему функции.

```
#include "TXLib.h"

void DrawHouse1();
void DrawHouse2();
void DrawHouse3();

int main()
    {
    txCreateWindow (500, 500);
```

```
    DrawHouse1();
    DrawHouse2();
    DrawHouse3();

    return 0;
    }

void DrawHouse1()
    {
    txLine (100, 150, 150, 100);
    txLine (200, 150, 150, 100);
    txLine (100, 150, 200, 150);
    txLine (200, 150, 200, 250);
    txLine (200, 250, 100, 250);
    txLine (100, 250, 100, 150);
    }

void DrawHouse2()
    {
    txLine (100, 350, 150, 300);
    txLine (200, 350, 150, 300);
    txLine (100, 350, 200, 350);
    txLine (200, 350, 200, 450);
    txLine (200, 450, 100, 450);
    txLine (100, 450, 100, 350);
    }

void DrawHouse3()
    {
    txLine (250, 250, 350, 150);
    txLine (450, 250, 350, 150);
    txLine (250, 250, 450, 250);
    txLine (450, 250, 450, 450);
    txLine (450, 450, 250, 450);
    txLine (250, 450, 250, 250);
    }
```

Снова ох. Три домика – функции. Почти полсотни строк.

Смотрите, косвенный формальный признак – количество строчек с txLine не сократилось. Тест на масштабирование программа проходит тоже очень плохо: десять домиков – десять функций DrawHouse, десять вызовов, десять прототипов, более 160 строк. Ужасно.

А если после отладки программы понадобится внести изменение в домики? Например, пририсовать дверь (добавить три линии в каждый домик), или сделать пониже крышу (изменить две линии)?

Ключевые слова здесь "в каждый домик". В каждый. А их три, десять, тридцать. Это каторга, а не работа. Копипаст страшен не вначале, он коварный. Он проявляет свою гадскую сущность потом, когда приходится менять код, и тогда обязательно забудешь поменять во всех скопированных местах. Практически невозможно постоянно держать все участки скопированного кода в соответствии друг с другом. Это адская работа.

На этой стадии не любящий трудиться работу бросает, не любящий думать – продолжает клепать домики методом копипаста (Ctrl+C / Ctrl+V), а разумный человек все же задумается.

Фактически, на рисунке один и тот же объект, но в трех экземплярах. Рассуждая так, мы немного обобщаем задачу его рисования. Ведущий способ обобщения – параметры функции, с помощью которых мы задаем конкретные характеристики конкретных объектов (домиков), которые мы хотим видеть на экране. Если так рассудить, то получится уже лучше:

```
#include "TXLib.h"

void DrawSmallHouse (int x, int y);
void DrawBigHouse   (int x, int y);

int main()
    {
    txCreateWindow (500, 500);

    DrawSmallHouse (100, 150);
    DrawSmallHouse (100, 350);
    DrawBigHouse   (250, 250);

    return 0;
    }

void DrawSmallHouse (int x, int y)
    {
    txLine (x,       y, x + 50,  y - 50);
    txLine (x + 100, y, x + 50,  y - 50);

    txLine (x,       y,       x + 100, y);
    txLine (x + 100, y,       x + 100, y + 100);
    txLine (x + 100, y + 100, x,       y + 100);
    txLine (x,       y + 100, x,       y);
    }

void DrawBigHouse (int x, int y)
    {
    txLine (x,       y, x + 100, y - 100);
    txLine (x + 200, y, x + 100, y - 100);

    txLine (x,       y,       x + 200, y);
    txLine (x + 200, y,       x + 200, y + 200);
    txLine (x + 200, y + 200, x,       y + 200);
    txLine (x,       y + 200, x,       y);
    }
```

Вот, кстати, смотрите, здесь благодаря вертикальному выравниванию за счет дополнительных пробелов легче следить за логикой функции, и перепутать, например, икс с игреком уже не удастся. И рисование крыши отделено пустой строкой от остального дома для понятности.

Эта программа лучше, потому что мы можем дорисовать еще десяток домиков всего десятью дополнительными строками – вызовами функции DrawSmallHouse или DrawBigHouse. Она хорошо масштабируется по количеству домиков, и написать ее проще – не надо дважды вручную прорисовывать маленький домик. Но по размеру домиков она масштабируется плохо: если нам нужны домики других размеров, придется писать дополнительные функции.

Решение опять в обобщении, вспомним, что мы говорили об одном объекте – домике, а не о двух. У нас фактически один домик, но разных размеров. Если размер мы зададим как параметр функции size (размер), то она потребуется всего одна, плюс три ее вызова:

```
#include "TXLib.h"

void DrawHouse (int x, int y, double size);

int main()
    {
    txCreateWindow (500, 500);

    DrawHouse (100, 150, 1);
    DrawHouse (100, 350, 1);
    DrawHouse (250, 250, 2);

    return 0;
    }

void DrawHouse (int x, int y, double size)
    {
    txLine (x,            y, x + 50*size,  y - 50*size);
    txLine (x + 100*size, y, x + 50*size,  y - 50*size);

    txLine (x,            y,            x + 100*size, y);
    txLine (x + 100*size, y,            x + 100*size, y + 100*size);
    txLine (x + 100*size, y + 100*size, x,            y + 100*size);
    txLine (x,            y + 100*size, x,            y);
    }
```

или, если размер задавать как абсолютное число, а не относительное:

```
#include "TXLib.h"

void DrawHouse (int x, int y, int size);

int main()
    {
    txCreateWindow (500, 500);

    DrawHouse (100, 150, 100);
    DrawHouse (100, 350, 100);
    DrawHouse (250, 250, 200);

    return 0;
    }

void DrawHouse (int x, int y, int size)
    {
    txLine (x,        y, x + size/2, y - size/2);
    txLine (x + size, y, x + size/2, y - size/2);

    txLine (x,        y,        x + size, y);
    txLine (x + size, y,        x + size, y + size);
    txLine (x + size, y + size, x,        y + size);
```

```
    txLine (x,        y + size, x,        y);
    }
```

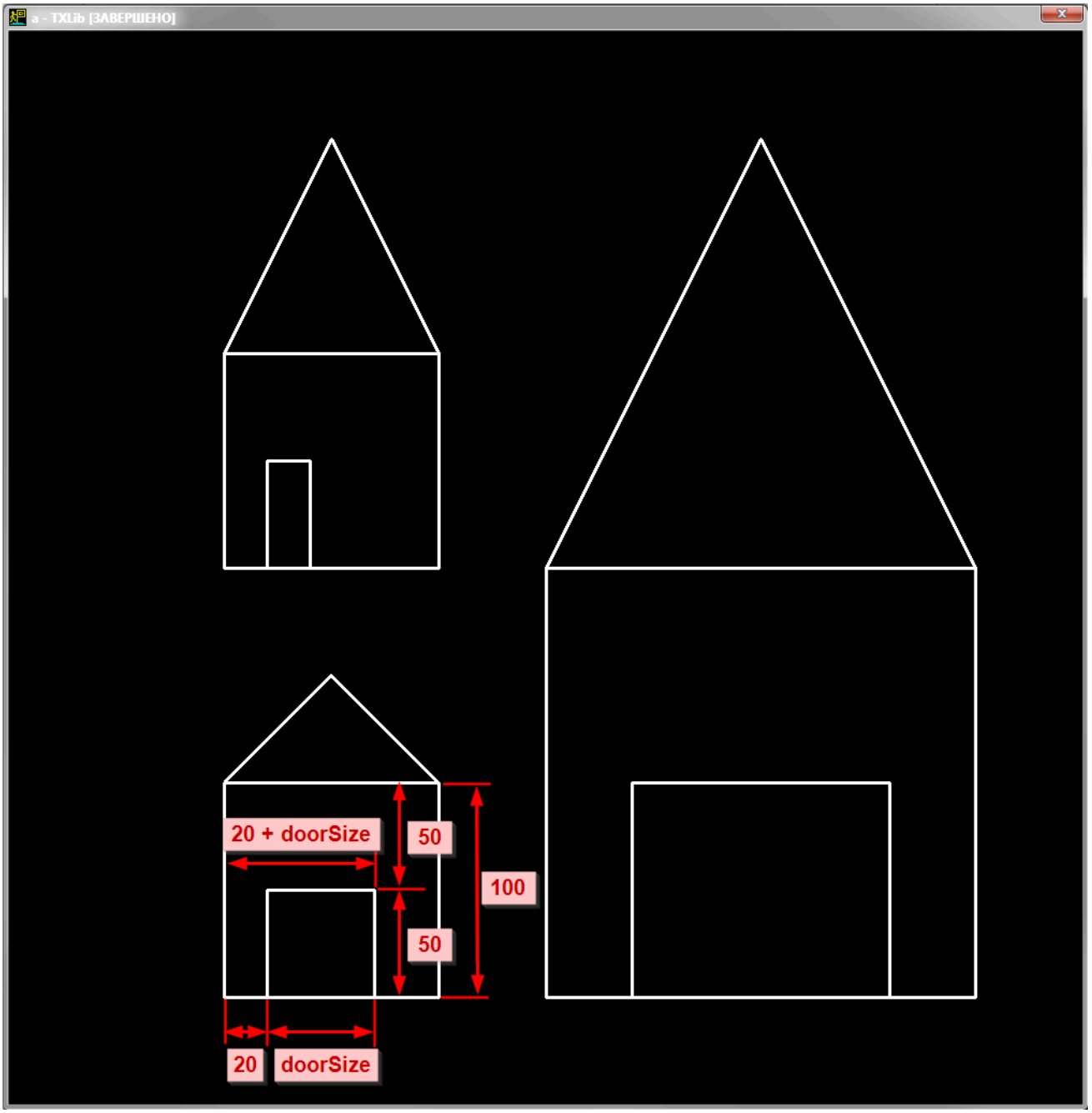
В этом случае рисуем три домика оптимальным образом, и это решение отлично масштабируется.

Хотим увеличить высоту крыши и пририсовать дверь? Нет проблем:

```
#include "TXLib.h"

void DrawHouse (int x, int y, double size, int roofSize, int doorSize);

int main()
    {
    txCreateWindow (500, 500);

    DrawHouse (100, 150, 1, 50, 20);
    DrawHouse (100, 350, 1, 25, 50);
    DrawHouse (250, 250, 2, 50, 60);

    return 0;
    }

void DrawHouse (int x, int y, double size, int roofSize, int doorSize)
    {
    txLine (x,            y, x + 50*size, y - roofSize*size);
    txLine (x + 100*size, y, x + 50*size, y - roofSize*size);

    txLine (x,            y,            x + 100*size, y);
    txLine (x + 100*size, y,            x + 100*size, y + 100*size);
    txLine (x + 100*size, y + 100*size, x,            y + 100*size);
    txLine (x,            y + 100*size, x,            y);

    txLine (x +  20             * size, y + 100*size, x +  20             * size, y + 50 *size);
    txLine (x +  20             * size, y + 50 *size, x + (20 + doorSize) * size, y + 50 *size);
    txLine (x + (20 + doorSize) * size, y + 50 *size, x + (20 + doorSize) * size, y + 100*size);
    }
```

На рисунке ниже, полученном с помощью этой программы, показан расчет координат двери и происхождение компонента (20 + doorSize) в коде выше.

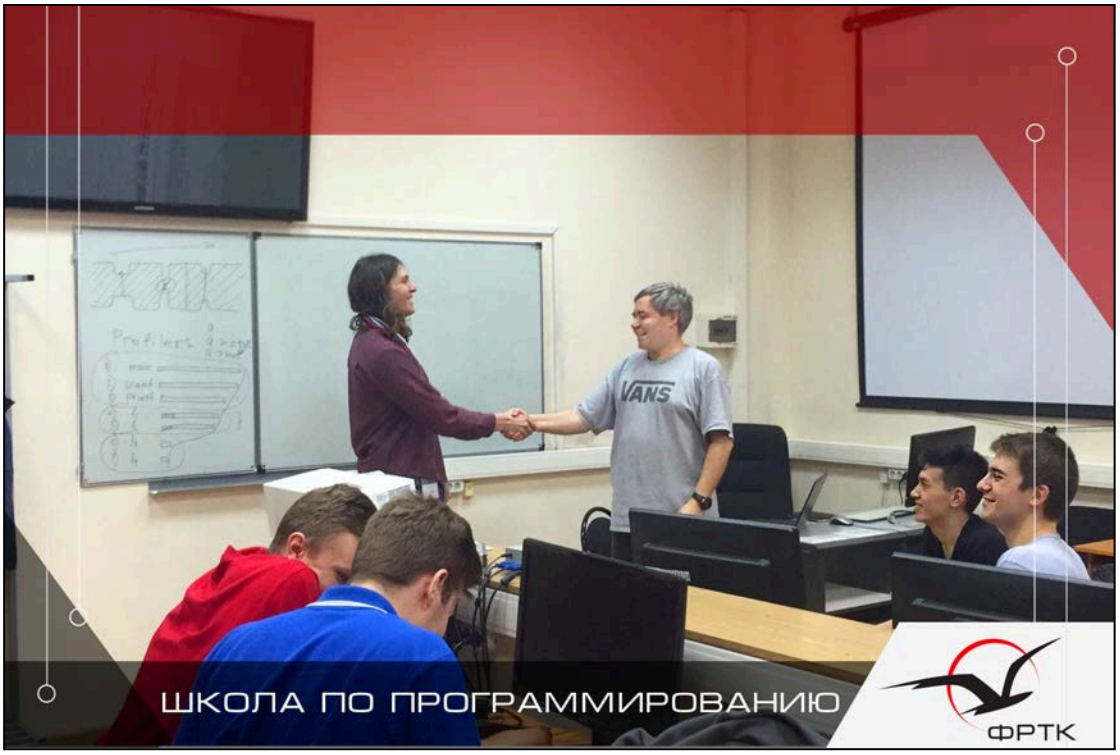
Совсем хорошо было бы задать опорную точку дома (координаты x, y) не под крышей, а, например, на середине его основания. Тогда было бы легко расставлять на одном уровне несколько домов разного размера. Это немного усложнит расчеты, но это стоит того. Понятное дело, что в контрольную для начинающих такие вещи уже войдут только "со звездочкой", как желательное, но необязательное задание. Но вне контрольной стоит предпочитать логичную привязку объектов, так как функцию мы пишем один раз, а пользуемся ей много раз, и надо упростить, оптимизировать и сделать удобным именно ее пользование.



Может быть, многое при рассмотрении этого примера было вам очевидно, и хорошо, если так. Но вы не представляете, насколько часто встречаются подобные ошибки, конечно, не в таком явном виде, а в более завуалированном. Этим грешат школьники, студенты и даже начинающие программисты. В основе этих ошибок лежит привычка к копированию кода, без мыслей о масштабировании и обобщении. Упомянутый выше принцип DRY предписывает не делать так: DRY – это сокращение от Don’t Repeat Yourself, “не повторяй сам себя”. То есть не копируй, не копипасть. Следовать ему надо неукоснительно. Вместе с “разделяй и властвуй” он составляет фундамент базовых принципов архитектуры кода.

Автор не может не рассказать забавный случай, произошедший с одним из его студентов-первокурсников. Весь семестр его продвинутая группа усердно переучивалась с разных знаний по программированию, почерпнутых в школе, на профессиональные стандарты. Конечно, не на простых графических задачах: увы, так плавно строить курсы в вузе не удается, там нужен высокий темп. Пришло время зачета, каждый получил по задаче и принялся решать. От Леши, знающего программирование еще со школы, автор ожидал довольно быстрого решения. Не тут-то было: Леша все написал, но искал ошибку. Сопел, рычал (а когда Леша рычит – это всегда громко, у него еще со школы прозвище Чубакка),

задача не поддавалась. Дело шло к концу зачета, автор уж думал, что придется зачитывать задачу частично, прощай, отл (10)[2]. И вдруг из недр Леши раздался даже не рык, а ураганный вопль: "ЭТО КОПИПАААААСТ!!! ...аааст... аааст... аааст..." Звук вырвался в открытую дверь и волной пошел гулять по коридору. Заглядывали испуганные лаборанты... (помним про Лешино прозвище). Оказалось, что ради простоты Леша несколько раз скопировал кусок кода задачи, и в каждой из копий, естественно, что-то немного поменял. Потом, спустя время, понадобилось поменять еще раз, и вот тут Леша поменял уже не везде, ибо замена была хитрой, а сколько раз копипастил – он забыл. Одно из скопированных мест он пропустил. В результате программа непредсказуемо падала, причем не всегда, а лишь иногда – когда исполнялся тот самый, забытый копипастный код. После дикого крика Леша лихорадочно поправил пару строк, до конца зачета он успел и "отл" свой получил, но вредный автор, зная, что Леша увлекается 3D-печатью, в шутку обещал наградить его медалями "за копипаст", который Леша распечатал бы на 3D-принтере. В начале следующего семестра произошло торжественное награждение:



Вот эта легендарная награда:

[2] На Физтехе десятибалльная система оценок, дублирующаяся словами, от неуд (1) до отл (10).

(Не смотрите, что на медали полосы, это бракованная версия, которую автор выпросил у Леши показывать другим студентам и рассказывать об этом случае. Леша был награжден нормальной версией. Бракованная же версия верой и правдой долго служит автору, когда он рассказывает эту историю и демонстрирует медали.)

Обратите внимание: медаль парная, обе части носятся обязательно вместе, носить одну медаль вместо двух – неуважение к награде, это запрещено.

Такие вот дела. Леша уж выпустился давно, а сказ про эпичный копипаст живет в веках. Вы, кстати, тоже можете напечатать и награждать, если есть 3D-принтер. Наверняка ведь будет кого награждать. :) Ну или грамоты можно выдавать. Но медаль почетнее.

1.4. Повторение действий (циклы)

Окей, с функциями, рисующими героев, мы разобрались. Но что делать с движением?

Движение проще всего делать через перерисовку. Самая простая прорисовка – когда на экране перерисовываются все фигуры, начиная с фона. На современных компьютерах это не занимает много времени. Если повторять перерисовку, изменяя положение героев, то получим их движение.

Рассмотрим движение человечка по горизонтали. Для этого, конечно, сделаем функцию, MoveMan, например.

Как записать в программе повторение? Конечно, можно вручную, через тщательный набор каждой повторяемой строчки вручную (мы же не пользуемся Ctrl+C + Ctrl+V, да?:)). Но, само собой, это нерационально. Давайте попробуем выразить мысль о повторении на *псевдокоде* – это запись на русском/английском/другом языке, но предельно конкретная и очень близкая к программе. Использование псевдокода позволяет отделить продумывание логики программы от записи ее на языке программирования по правилам его синтаксиса. Это очень полезно в случаях, когда логика непроста, или когда язык программирования не выучен целиком и его синтаксис не всегда ясен.

Получается так:

```
Алгоритм движения человечка

Пусть координата x вначале будет равна 100

Пока эта координата x меньше или равна 700, делай:
    (Начало повтора)
    Сотри все с экрана
    Нарисуй человечка в точке с координатами x, 200
    Увеличь координату x на 10
    Сделай небольшую паузу
    (Конец повтора)
```

Или, сокращенно,

```
Алгоритм движения человечка
```

```
Целое число x = 100

Пока x <= 700:
    (Начало повтора)
    Сотри все с экрана
    Нарисуй человечка в точке (x, 200)
    Увеличь x на 10
    Сделай небольшую паузу
    (Конец повтора)
```

Если показать такой псевдокод программисту, он быстро поймет, в чем дело, и может легко подсказать конкретный синтаксис для записи на конкретном языке программирования. Если же подойти к нему с расплывчатыми вопросами, то скорее всего особой помощи не будет. Мораль: логику пишите на псевдокоде, можно русском, можно английском, можно на псевдо-сишном, неважно; обсуждайте ее с другими, показывайте знающим людям, потом переводите его в код на нормальный Си/C++ или другие языки программирования.

Обратите внимание, мы завели переменную, отвечающую за x-координату человечка, но она не является параметром функции MoveMan. Совсем не обязательно пользоваться переменными-параметрами, всегда можно завести *локальную переменную,* это как промежуточная переменная, не входящая в условие, при решении сложной задачи по физике, например. Вы ведь имеете право всегда завести такую переменную, если вам удобно? Вот и в Си так можно.

Один из операторов, реализующих повторение действий в Си – это *while.* Вот что получится после перевода псевдокода с его использованием:

```
void MoveMan()
    {
    int x = 100;

    while (x <= 700)
        {
        txSetFillColor (TX_BLACK);
        txClear();

        DrawMan (x, 200);
        x += 10;            // Увеличь x на 10

        txSleep (100);      // Сделай небольшую паузу (на 100 миллисекунд)
        }
    }
```

Тут надо быть внимательным. С циклами легко сделать ошибки. Смотрите, вот этот цикл ни разу не выполнится:

```
void MoveMan()
    {
    int x = 100;

    while (x >= 700)
        {
        txSetFillColor (TX_BLACK);
        txClear();

        DrawMan (x, 200);
```

```
        x += 10;

        txSleep (100);
        }
    }
```

Заметили? Условие выполнения цикла сразу ложно, и поэтому ни одна строчка тела цикла не выполнится ни разу. Это, скорее всего, обычная опечатка.

Вот еще цикл с чуть более хитрой ошибкой:

```
void MoveMan()
    {
    int x = 100;

    while (x <= 700)
        {
        txSetFillColor (TX_BLACK);
        txClear();

        DrawMan (x, 200);
        x -= 10;

        txSleep (100);
        }
    }
```

Этот цикл выполнится, но никогда не завершится, как говорят, он "зависнет", станет бесконечным. Это потому, что условие будет всегда истинным, а это будет потому, что (заметили?) координата x уменьшается, а не увеличивается. (На практике, через некоторое время возникнет *переполнение при вычитании,* и координата резко станет очень большой и цикл завершится, но об этом сейчас не будем, тем более, что все равно это будет ошибочная ситуация.)

Более простую ошибку можно создать, если вообще забыть написать x -= 10; или аналогичную конструкцию. В этом случае переменная x вообще не будет меняться, и цикл опять зависнет.

Когда эти ошибки найдены, они кажутся очевидными. Но до этого найти их иногда непросто. В этом может помочь *отладочная распечатка* значений переменных (как говорят программисты, *дамп*). Сделать ее очень легко:

```
void MoveMan()
    {
    int x = 100;

    while (x <= 700)
        {
        txSetFillColor (TX_BLACK);
        txClear();

        printf ("MoveMan(): x = %d\n", x);

        DrawMan (x, 200);
        x -= 10;

        txSleep (100);
        }
    }
```

На экране вместе с рисунком появятся строки:

```
MoveMan(): x = 100
MoveMan(): x = 90
MoveMan(): x = 80
MoveMan(): x = 70
MoveMan(): x = 60
MoveMan(): x = 50
MoveMan(): x = 40
MoveMan(): x = 30
MoveMan(): x = 20
MoveMan(): x = 10
MoveMan(): x = 0
MoveMan(): x = -10
...
```

Видите – x уменьшается, и становится даже отрицательным числом. В этом месте проблема становится очевидна, и можно думать о ее решении. Почему очевидна? Потому что мы *сделали невидимое видимым* с помощью *логгирования (логгинга, иногда еще говорят "журналирование" – от слов "log", "журнал")*, и в этом один из мощных приемов отладки программ. Спросите любого программиста про *логи,* он будет знать, что это такое. В простейшем виде логгинг – это просто вывод на экран информации, но главное, чтобы побольше и чтобы всем было понятно, а не только автору программы. :) Тогда, кстати, друзья и коллеги смогут свежим взглядом посмотреть, быстро все понять и найти ошибку быстрее, чем автор. Вполне себе рабочий способ при групповых проектах, да и просто при совместном изучении программирования.

Эту распечатку выводит функция printf – стандартная функция языка Си, она есть на всех платформах и предназначена для распечатки, или вывода данных в консоль программы. В библиотеке TXLib консоль совмещена с графическим окном программы, и распечатки появляются поверх рисунков. (Если хочется, можно показать отдельное консольное окно, щелкнув правой кнопкой мыши на заголовке окна и выбрав "Show Console".) Функция printf выводит на экран строку с текстом, позволяя подставлять в нее значения переменных в местах, указанных символом процента "%". После символа процента нужно указывать, какого типа значение нужно вывести: %d – для вывода целых чисел, %lg – для вывода дробных чисел. Для вывода символов используется %c, строк - %s. Что именно подставлять вместо таких "процентов", указывается после строки отдельными параметрами, у нас это переменная x, а вообще переменных может быть много. Символ \n означает переход на новую строку, чтобы распечатка шла столбиком. (Найдите информацию также о других специальных символах языка Си, здесь она слишком краткая. Также найдите информацию о выводе разных типов данных в документации на функцию printf.) Вот как схематически работает эта подстановка:

```
int x = 100, y = 200;
...

//                      +--------------+
//                      |              |
//                      V              |
printf ("MoveMan(): x = %d, y = %d\n", x, y);
//                                ^         |
//                                |         |
//                                +---------+
```

И в результате выводится

```
MoveMan(): x = 100, y = 200
```

Не путайте обозначения форматов, потому что тогда printf выдаст ерунду, вы в нее поверите и провозитесь с отладкой значительно дольше. То же самое относится к соответствию количества "процентов" и подставляемых в них выражений. Например, такие распечатки выдадут ерунду:

```
int x = 100;
printf ("x = %lg\n", x);   // Переменная x - целочисленная (int), а %lg - это вывод дробных чисел (double)

double y = 10;
printf ("y = %d\n", y);    // Переменная y - типа double, а %d - это вывод целых чисел (int)
printf ("x * 0.5 = %d\n", x * 0.5); // А вот тут проблема тоньше. Переменная x целочисленна (int), но число
                                    // 0.5 имеет тип double, потому что имеет дробную часть. Поэтому произведение,
                                    // чтобы не потерять точности, тоже имеет тип double и должно
выводиться
                                    // через формат %lg.

printf ("x = %d\n", x, y);          // В строке вывода один "процент", а в списке подставляемых значений - две
                                    // переменные. Вторая переменная не будет распечатана.

printf ("x = %d, y = %lg\n", x);    // В строке вывода два "процента", а в списке подставляемых значений - одна
                                    // переменная. Вместо второй переменной выведется ерунда.
```

Проверьте, кстати, что именно выводят эти распечатки, чтобы вы дальше по интуиции распознавали такие ошибки. Согласитесь, ошибка в распечатке наиболее обидна: это ведь не ошибка в логике программы, но вы эту ошибку ищете, потому что верите распечатке, что в программе есть проблема. :) (Здесь автор вспоминает о многих потерянных часах из-за таких ошибок.)

Хорошо настроенный компилятор языка Си может выдавать предупреждения о таком несоответствии – и по количеству, и по типам. Если вы пользуетесь библиотекой TXLib, то она уже настраивает компилятор таким образом с помощью указанной в библиотеке директивы

```
#pragma GCC diagnostic warning "-Wformat=2"
```

Либо можно вызывать компилятор из командной строки с опцией `-Wformat=2`:

```
g++ -Wformat=2 Program.cpp
```

В этом случае вы получите предупреждения вида

```
warning: format '%lg' expects argument of type 'double', but argument 2 has type 'int' [-Wformat=]
warning: format '%d' expects argument of type 'int', but argument 2 has type 'double' [-Wformat=]
warning: too many arguments for format [-Wformat-extra-args]
warning: format '%lg' expects a matching 'double' argument [-Wformat=]
```

Переведите эти предупреждения с английского и посмотрите, каким строкам и ошибочным случаям они соответствуют.

Еще вот такой момент. Смотрите, как построена распечатка. В ней указано имя функции, с которой мы работаем, название переменной и ее значение. Не надо пренебрегать названием переменной и другим поясняющим текстом. Почему? Давайте промасштабируем ситуацию, когда мы выводим на экран только значение переменной в виде числа, как часто делают неопытные начинающие. Тогда на экране будут десятки чисел, каждое из которых что-то значит, и в голове придется держать их смысл. Даже если это у вас и получится (привычные к такому подходу люди не замечают этого напряжения мозгов), вашего внимания уже не хватит

на действительно сложную задачу, и вы будете решать ее дольше, путаясь. Зачем это? Почему не отлаживать комфортно, попивая вкусный чай и не утруждая себя запоминанием того, что может вам напомнить ваша же программа? Отладка и так муторное дело, не надо ее превращать в еще большее мучение. Просто напишите чуть больше информации, и многие вещи станут понятнее сами собой – как вам, так и вам же, спустя неделю (поверьте, это два совсем разных человека:)), так и друзьям-коллегам, с которыми вы вместе учитесь-работаете.

Почему мы так много говорим об ошибках? Потому что они будут. :) Программирование невозможно без ошибок, к ним надо относиться нормально и привычно, не комплексовать и не тильтовать, как говорится. Кажется, что профессия программиста безэмоциональна – пиши код, и всё – но это не так: каждый день ты ощущаешь себя то гением, то идиотом. :) Чем опытнее программист, тем лучше он знает, что ошибок полностью избежать невозможно, и одно из искусств программирования состоит в том, чтобы перевести возникновение ошибки на более раннюю стадию исполнения программы (лучше, чтобы прямо сразу после запуска – есть такая технология "юнит-тестирование"). Для этого даже часто пишут специальные функции, тестирующие части программы до того, как пользователь начнет с ней работать. Конечно, лучше всего перевести детекцию ошибки на стадию компиляции, но это не всегда возможно.

Окей, теперь мы можем двигать человечка по горизонтали. Можем мы его двигать одновременно и по вертикали? Можно, один из способов – это, очевидно, завести локальную переменную y. Ее совсем не обязательно вставлять в условие цикла, там достаточно условия с переменной x, а усложнять условия без необходимости – риск порезаться бритвой Оккама. :) Но не забудьте изменять эту переменную в цикле, а то мы ее тогда, получается, зря заводили.

Можно ли задавать координаты не целыми числами? Несмотря на то, что координаты пикселей картинки целочисленные, для задания движения удобно использовать вещественные числа (double), чтобы точнее задавать скорости движения. Заметим, что если координата x целочисленна, то код

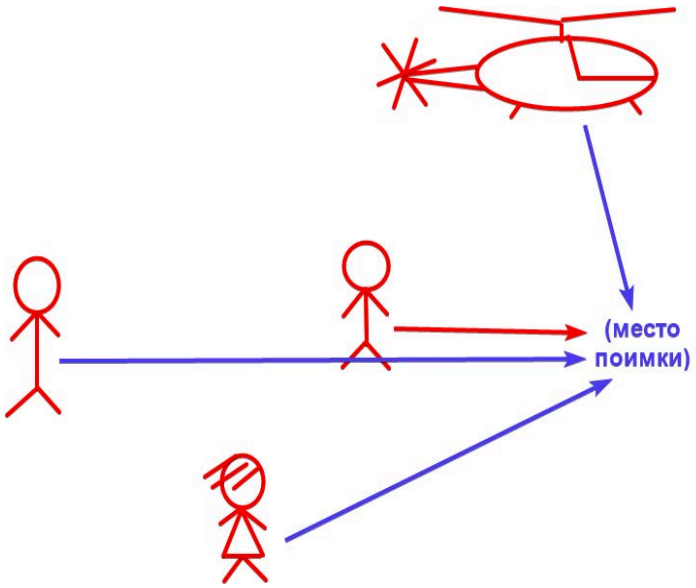
```
int x = 100;
...
x += 0.1;
```

никак не изменит переменную x, потому что после добавления 0.1 у нее отбросится дробная часть, так как x объявлена как целочисленная переменная. Сюрприз, будьте внимательны. А код

```
double x = 100;
...
x += 0.1;
```

сработает правильно.

Окей. Как делать движение одного героя, более-менее понятно. Чтобы было совсем понятно, остановитесь и поэкспериментируйте с движением своих героев. Теперь давайте промасштабируем этот подход. Пусть у нас есть сцена погони, где двое полицейских бегут за преступником, а сверху еще и снижается полицейский вертолет. (В Pixar нам стало скучно, мы вслед за Брэдом Бердом ушли оттуда снимать боевик – а Мосфильм все еще тянет с приглашением.) Для каждого объекта есть свои координаты x и y, всего объектов четыре, переменных всего уже восемь. Вот как мог бы выглядеть код погони в стиле с отдельными переменными для каждого объекта:

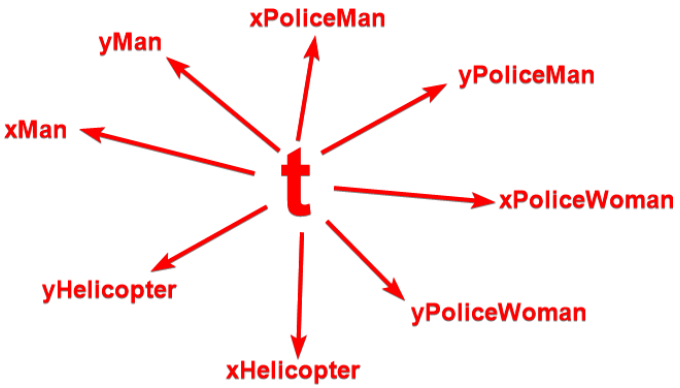



```
void PoliceChase()
    {
    double xMan = 400, yMan = 250;
    double xPoliceMan = 50, yPoliceMan = 270;
    double xPoliceWoman = 250, yPoliceWoman = 400;
    double xHelicopter = 600, yHelicopter = 0;

    while (xMan <= 700)
        {
        txSetFillColor (TX_BLACK);
        txClear();

        DrawMan (xMan, yMan);
        DrawMan (xPoliceMan, yPoliceMan);
        DrawWoman (xPoliceWoman, yPoliceWoman);
        DrawHelicopter (xHelicopter, yHelicopter);

        xMan         += 5;
        yMan         += 1;
        xPoliceMan   += 10;
        yPoliceMan   += 1;
        xPoliceWoman += 7;
        yPoliceWoman -= 0.35;
        xHelicopter  += 1;
        yHelicopter  += 2.5;

        txSleep (100);
        }
    }
```

Бррр. Такое себе. :(А если у нас больше объектов? Ясно, это плохо масштабируется.

Давайте подумаем.

Да, у нас четыре объекта и восемь переменных. Но, может быть, есть что-то, от чего они все зависят? Что-то их объединяющее? Физика говорит нам, что у нас задача одновременного движения четырех тел. Их объединяет… (тут подумайте, не читайте дальше!) …время, прошедшее с начала движения. То есть вот так:



Получается, все восемь координат можно рассчитать из одной-единственной переменной t по закону равномерного прямолинейного движения, зная координаты начала движения точек и их скорости по осям x и y. Откуда они нам известны? А мы их посчитаем. Все равно, чтобы добиться нужного кинематографического эффекта, придется менять параметры движения, это нормально. Примем, что движение займет у нас 100 условных секунд или кадров (на самом деле – оборотов цикла), тогда начале движения время t = 0, а в конце t = 100. С числом 100 проще работать, так-то можно взять любое разумное. Пусть преступник начинает свое движение с точки (x = 400, y = 250) и заканчивает свое движение в точке (x = 700, y = 310). Тогда его скорости будут равны v_x = (700 - 400) / 100 = 3 и v_y = (310 - 250) / 100 = 0.6. Если эти скорости нас не устраивают, можно потом подкорректировать.

Конечно, если бы у нас все данные о движении были бы известны заранее, то мы бы просто указали их в программе, это было бы даже легче. Со строго заданными условиями всегда проще работать. Но, к сожалению, хорошо поставленные задачи нечасто встречаются на практике.

Нужно ли по-прежнему заводить восемь переменных для координат объектов? Можно уже не заводить. Хотя, если с ними понятней, можно и завести. Если оставить только формулы движения (вычисление текущих координат объектов в зависимости от времени), получится вот что:

```
void PoliceChase()
    {
    double t = 0;
    while (t <= 100)
        {
        txSetFillColor (TX_BLACK);
        txClear();

        DrawMan        (400 +   3*t, 250 + 0.6*t);
        DrawMan        ( 50 +   6*t, 270 + 0.6*t);
        DrawWoman      (250 +   4*t, 400 - 0.3*t);
        DrawHelicopter (600 + 0.5*t,   0 + 1.5*t);

        t++;  // Операция увеличения на 1 (инкремент). Это то же самое, что t = t+1 или t += 1

        txSleep (100);
        }
    }
```

Гораздо легче, чем через отдельную работу с каждой из восьми переменных. Тем более, что преступник пойман, увезен на вертолете и можно насладиться свежеполученным Оскаром за лучшую режиссуру и спецэффекты.

1.4.1. Непрямолинейное движение

Можно ли сделать движение непрямолинейным? Конечно, можно.

Например, нам нужно чтобы у человечка двигались ноги, туда-сюда, то есть нам нужно возвратно-поступательное движение. Если бы было нужно просто поступательное, то координата, скажем, правой ноги простейшим образом зависела бы от времени так:

```
t    = 0 1 2 3 4 5 6 7 8 9 10 ...
rLeg = 0 1 2 3 4 5 6 7 8 9 10 ...
```

Но нам нужно, скажем, так (тоже простейшим образом):

```
t    = 0 1 2 3 4 5 6 7 8 9 10 ...
rLeg = 0 1 0 1 0 1 0 1 0 1 0  ...
```

Как это получить? Какая арифметическая операция дает 0 на четных числах и 1 на нечетных?

...Четные числа делятся на 2 нацело, нечетные не делятся нацело...

...Если число не делится нацело, то оно делится с остатком...

...Если нечетное число делится на 2 с остатком, то в этом остатке будет 1...

...В таблице у нас напротив нечетных чисел стоят как раз единицы...

Нам нужна операция получения остатка от деления. На число 2.

Ок. Как она обозначается? Гуглим (а лучше начинаем читать систематический учебник по Си – рекомендую книгу: Стивен Прата, “Язык Си, лекции и упражнения”, последнее издание. (Да, она толстая. Прочитаете. Вы на Физтех собрались поступать, неужели не прочитаете? Да, там нет проектов. Но для проектов есть эта книжка). Обозначается она символом процента: %. То есть rLeg = t % 2.

Отлично.

```
t     = 0 1 2 3 4 5 6 7 8 9 10 ...
t % 2 = 0 1 0 1 0 1 0 1 0 1 0  ...
```

Так. Но амплитуда (диапазон значений t % 2), равная единице – это мало. Мы особо ничего не увидим, нога будет двигаться слабо. Как увеличить амплитуду? Это эквивалентно вопросу

```
 t % 2          = 0 1 0 1 0 1 0 1 0 1 0  ... как превратить этот ряд
(t % 2) ...???  = 0 5 0 5 0 5 0 5 0 5 0  ... например, в этот
```

Очевидно, каждое число надо домножить на 5. 5 это тоже мало, домножьте сами на что-то большее.

Так. А если надо не 0..5, а 10..15?

```
(t % 2) * 5         =  0  5  0  5  0  5  0  5  0  5  0  ...
(t % 2) * 5 ...???  = 10 15 10 15 10 15 10 15 10 15 10  ...
```

Подумайте, какая арифметическая операция дает такую последовательность.

Ок. А если надо так:

```
t     = 0 1 2 3 4 5 6 7 8 9 10 11 ...
t...? = 0 1 2 0 1 2 0 1 2 0 1  2  ...
```

Что сбрасывается до нуля на каждом третьем числе? Какое это имеет отношение к делимости на 3 и как это записать какой-то операцией на Си? Подумайте.

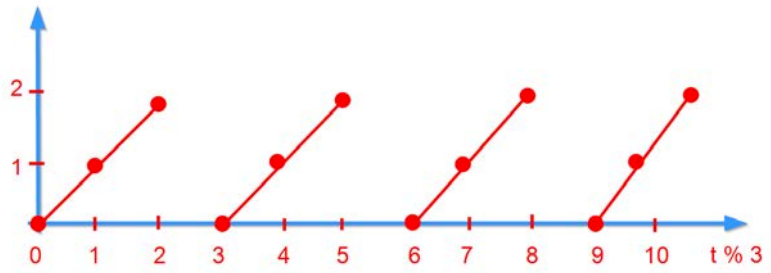
А вот такое?

```
t     = 0 1 2 3 4 5 6 7 8 9 10 11 ...
t...? = 0 0 0 1 1 1 2 2 2 3 3  3  ...
```

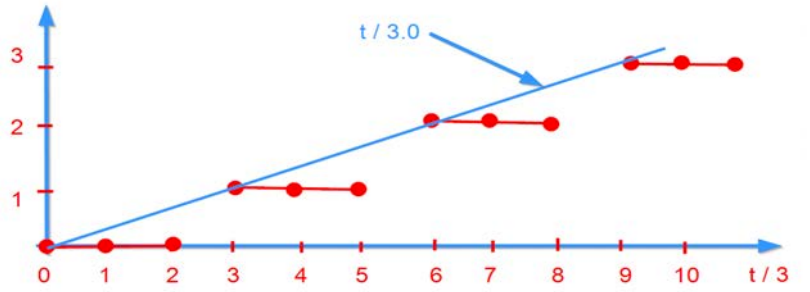
Здесь результат операции растет в 3 раза медленнее переменной t. Значит, надо поделить t на что-то. Только целочисленно, а то будут дробные части. Но арифметические операции с целыми числами в Си дают именно целые числа, и деление целых чисел происходит с отбрасыванием остатка. Поэтому 4/3 = 1. А вот 4.0/3.0 = 4.0 / 3 = 4 / 3.0 = 1.333... Вы уже начали систематически читать книжку Праты? В соответствующем месте там про это подробно написано.

Все вышесказанное, кстати, имеет прямое отношение к теме по математике “Преобразование графиков на плоскости”. В самом деле, вот слева таблицы, а справа графики:

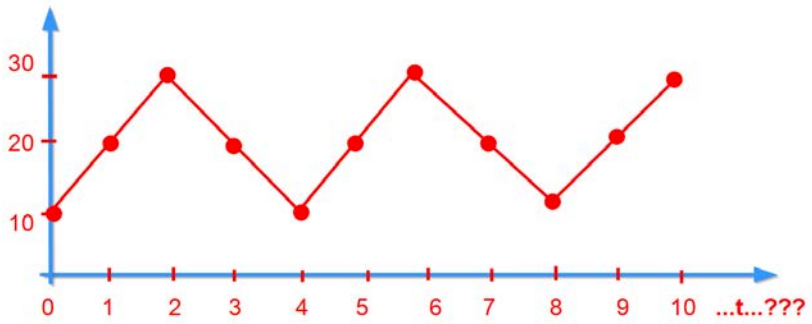
```
t     = 0 1 2 3 4 5 6 7 8 9 10 11 ...
t % 3 = 0 1 2 0 1 2 0 1 2 0 1  2  ...
```




```
t     = 0 1 2 3 4 5 6 7 8 9 10 11 ...
t / 3 = 0 0 0 1 1 1 2 2 2 3 3  3  ...
```



Только получается обратная задача – не “нам дана формула, постройте график”, а “дан график, подберите под него формулу”. Попробуйте подобрать формулу вот под такой график (ноги симметрично движутся туда-сюда относительно среднего значения 20):



Можно ли сделать колебательное движение плавным? Для этого можно использовать периодические функции, например синус: x = 400 + 3*t, y = 250 + 0.6*t + 20 * sin (t). Попробуйте. Правда, характеристики такого движения рассчитать сложнее, чем прямолинейного, но, может быть, проще, чем строить целочисленные функции (но вы все равно попробуйте, в программировании это неплохо помогает).

Вот пример кода, где герои движутся, и при этом параметры, отвечающие за положения частей героев, меняются туда-сюда, вызывая забавную, немного дерганую, картину:

```
void MoveTykva()
    {
    int t = 0;
    while (t <= 100)
        {
        txSetFillColor (SkyColor);
        txClear();

        DrawSun    (285 + (t % 5),     100 + (t % 10),     0);
        DrawYolka  ( 50,               300 + (t % 3) * 5,  1, 1);
        DrawDerevo (165,               250 + (t % 2) * 10, TX_GREEN, TX_RED, 1, 1);
        DrawYolka  (350,               320 + (t % 3) * 10, 0.6, 0.8);
        DrawDerevo (265 + (t % 2) * 5, 310,                TX_GREEN, TX_BROWN, 1.4, 0.5);

        DrawTykva  (258, 337, 1, 1, 0, 180, 0);

        DrawSliva  ( 55, 255, TX_LIGHTGREEN, TX_GREEN, 2.7, 20 + t%20,   0 - t*10, 0 + t%30);
        DrawSliva  (353, 240, TX_MAGENTA,    TX_GREEN, 3,   0  + t%20, 180 + t*20, 5 + t%30);

        DrawWay    (200, 378 + (t % 3));

        DrawTykva  (50 + (t * 3), 338 + (t % 2) * 10, 0.6, 0.4, 20, 0,  10);

        t++;

        txSleep (10);
        }
    }
```

В центральной части цикла значения первых двух параметров в функциях DrawSun, DrawYolka, DrawDerevo, DrawTykva (да, забавная смесь русских и английских слов, но пусть, весело), отвечающих за координаты x и y, специально выровнены столбиком, чтобы проще было следить за положениями героев. Такое же выравнивание сделано в двух смежных вызовах функции DrawSliva, но уже для всех параметров, чтобы было проще следить за взаимным

соответствием движений двух слив, зеленой и малиновой, у которых по-разному моргают глаза, поднимаются и опускаются руки, двигаются рты.

Скобки вокруг оператора остатка от деления (%) поставлены только для понятности, так как этот оператор имеет приоритет, одинаковый с оператором умножения и деления. Если они загромождают код, можно внутреннюю часть цикла написать в таком стиле:

```
    DrawSun    (285 + t % 5,     100 + t % 10,     0);
    DrawYolka  ( 50,             300 + t % 3 * 5,  1, 1);
    DrawDerevo (165,             250 + t % 2 * 10, TX_GREEN, TX_RED, 1, 1);
    DrawYolka  (350,             320 + t % 3 * 10, 0.6, 0.8);
    DrawDerevo (265 + t % 2 * 5, 310,              TX_GREEN, TX_BROWN, 1.4, 0.5);
```

Или так, если хочется показать порядок действий не скобками, а игрой с присутствием-отсутствием пробелов:

```
    DrawSun    (285 + t%5,     100 + t%10,     0);
    DrawYolka  ( 50,           300 + t%3 * 5,  1, 1);
    DrawDerevo (165,           250 + t%2 * 10, TX_GREEN, TX_RED, 1, 1);
    DrawYolka  (350,           320 + t%3 * 10, 0.6, 0.8);
    DrawDerevo (265 + t%2 * 5, 310,            TX_GREEN, TX_BROWN, 1.4, 0.5);
```

Выбирайте тот способ, который ближе вам, но и одновременно предельно ясен для всех ваших коллег, с кем вы работаете, контактируете или потенциально сможете это делать. Если сомневаетесь – ставьте пробелов побольше. Скобки тоже можно использовать, только не переусердствуйте, не надо делать много уровней х вложенности, код станет слишком сложным.

1.4.2. Пример из документации TXLib

Этот пример входит в состав примеров TXLib (Пример: Циклы). Разберитесь в нем. Обратите внимание на функцию JumpMan. Функция MoveMan устроена немного сложнее математически в плане расчета положения человечка (x, y). Полного стирания окна и рисования всех героев заново тут не производится, вместо этого объекты рисуются то цветом фона (черным), то цветом, отличным от фонового (желтым). Так можно делать только на простых фонах и объектах. В большинстве случаев лучше перерисовывать всю картинку заново.

```
#include <conio.h>
#include "TXLib.h"

//------------------------------------------------------------------------

void JumpMan     (int x, int y, int sizeX, int sizeY, double jump,
                  COLORREF color, COLORREF bkColor, int jumps, int delay);

void MoveMan     (int fromX, int fromY, int toX, int toY,
                  int sizeX, int sizeY, COLORREF color, COLORREF bkColor,
                  int time, int steps);

void DrawMan     (int x, int y, int sizeX, int sizeY, COLORREF color,
                  double hand = 0, double legs = 0, double head = 0, double twist = 0);

void AppearEarth (int x, int y, int sizeX, int sizeY, COLORREF from, COLORREF to,
                  int time, int steps);
```

```
void DrawEarth   (int x, int y, int sizeX, int sizeY, COLORREF color);

void AppearText  (int x, int y, const char* text, COLORREF from, COLORREF to,
                  int time, int steps);

void DrawHello   (int x, int y, const char* text, int size, COLORREF color);

void UnwindFlag  (int x, int y, int fromSizeX, int toSizeX, int sizeY,
                  COLORREF color, COLORREF bkColor, int time, int steps);

void DrawFlag    (int x, int y, int sizeX, int sizeY, COLORREF color, COLORREF bkColor);

void DrawFrame   (int sizeX, int sizeY, int size, COLORREF color);

//---------------------------------------------------------------------------

int main()
    {
    txCreateWindow (800, 600);
    txTextCursor (false);

    DrawFrame (800, 600, 10, TX_WHITE);

    txSelectFont ("Times New Roman", 60);
    txSetTextAlign (TA_CENTER);

    AppearText  (400, 480, "Hello, world!", TX_BLACK, TX_LIGHTGREEN, 5000, 100);
    AppearEarth (400, 300, 400, 300,        TX_BLACK, TX_LIGHTCYAN,  5000, 100);

    MoveMan    (20, 150, 385, 150, 20, 40,  TX_YELLOW, TX_BLACK,     3000, 100);
    JumpMan    (385, 150, 20, 40, 0.25,     TX_YELLOW, TX_BLACK,       10, 100);
    UnwindFlag (400, 150, 0, 40, 60,        TX_YELLOW, TX_BLACK,      500, 100);

    return 0;
    }

//---------------------------------------------------------------------------

void JumpMan (int x, int y, int sizeX, int sizeY, double jump,
              COLORREF color, COLORREF bkColor, int jumps, int delay)
    {
    DrawMan (x, y, sizeX, sizeY, TX_BLACK, 0, 0, 0, 0);

    txBegin();

    int i = 0;
    while (i < jumps)
        {
        DrawMan (x, y - (int) (i%2 * jump*10), sizeX, sizeY, color,
                 (i%2 - 0.5) * jump/3, 0, (i%2 - 0.5) * -jump/3, 0);

        txSleep (delay);

        DrawMan (x, y - (int) (i%2 * jump*10), sizeX, sizeY, bkColor,
                 (i%2 - 0.5) * jump/3, 0, (i%2 - 0.5) * -jump/3, 0);
```

```
        i++;
        }

    DrawMan (x, y - (int) (jumps%2 * jump*10), sizeX, sizeY, color,
            (jumps%2 - 0.5) * jump/3, 0, (jumps%2 - 0.5) * -jump/3, 0);

    txEnd();
    }

//---------------------------------------------------------------------------

void MoveMan (int fromX, int fromY, int toX, int toY,
              int sizeX, int sizeY, COLORREF color, COLORREF bkColor,
              int time, int steps)
    {
    txBegin();

    int i = 0;
    while (i <= steps)
        {
        int x = fromX + (toX - fromX) * i/steps,
            y = fromY + (toY - fromY) * i/steps;

        DrawMan (x, y - i%6, sizeX, sizeY, color,   i%3 * 0.02, i%3 * -0.1, i%3 * 0.1, 0);

        txSleep (time / steps);

        DrawMan (x, y - i%6, sizeX, sizeY, bkColor, i%3 * 0.02, i%3 * -0.1, i%3 * 0.1, 0);
        i++;
        }

    DrawMan (toX, toY, sizeX, sizeY, color, 0, 0, 0);

    txEnd();
    }

//---------------------------------------------------------------------------

void UnwindFlag (int x, int y, int fromSizeX, int toSizeX, int sizeY,
                 COLORREF color, COLORREF bkColor, int time, int steps)
    {
    txBegin();

    int i = 0;
    while (i <= steps)
        {
        int sizeX = fromSizeX + (toSizeX - fromSizeX) * i/steps;

        DrawFlag (x, y, sizeX, sizeY, color,   bkColor);

        txSleep  (time / steps);

        DrawFlag (x, y, sizeX, sizeY, bkColor, bkColor);
        i++;
        }
```

```
    DrawFlag (x, y, toSizeX, sizeY, color, bkColor);

    txEnd();
    }

//---------------------------------------------------------------------------

void DrawMan (int x, int y, int sizeX, int sizeY, COLORREF color,
              double hand, double legs, double head, double twist)
    {
    txSetColor     (color);
    txSetFillColor (color);

    txLine (x + twist*sizeX, y - 0.35*sizeY, x, y - 0.7*sizeY);

    txLine (x + twist*sizeX, y - 0.35*sizeY, x - (0.5 + legs) * sizeX, y);
    txLine (x + twist*sizeX, y - 0.35*sizeY, x + (0.5 + legs) * sizeX, y);

    txLine (x, y - 0.65*sizeY, x - sizeX/2,   y - 0.4*sizeY);
    txLine (x, y - 0.65*sizeY, x + sizeX/1.2, y - (0.7 + hand) * sizeY);

    txCircle (x, y - sizeY + (0.3 + head) * sizeX, 0.3*sizeX);
    }

//---------------------------------------------------------------------------

void DrawEarth (int x, int y, int sizeX, int sizeY, COLORREF color)
    {
    txSetColor (color);

    int r = sizeX/2;
    while (r >= 0)
        {
        txEllipse (x - r, y - sizeY/2, x + r, y + sizeY/2);
        r -= sizeX/9;
        }

    r = sizeY/2;
    while (r >= 0)
        {
        txEllipse (x - sizeX/2, y - r, x + sizeX/2, y + r);
        r -= sizeY/6;
        }

    txLine (x - sizeX/2, y, x + sizeX/2, y);
    }

//---------------------------------------------------------------------------

void AppearEarth (int x, int y, int sizeX, int sizeY, COLORREF from, COLORREF to,
                  int time, int steps)
    {
    int r0 = txExtractColor (from, TX_RED),   r1 = txExtractColor (to, TX_RED),
        g0 = txExtractColor (from, TX_GREEN), g1 = txExtractColor (to, TX_GREEN),
        b0 = txExtractColor (from, TX_BLUE),  b1 = txExtractColor (to, TX_BLUE);
```

```
    int i = 0;
    while (i <= steps)
        {
        int r = r0 + (r1 - r0) * i/steps,
            g = g0 + (g1 - g0) * i/steps,
            b = b0 + (b1 - b0) * i/steps;

        DrawEarth (x, y, sizeX, sizeY, RGB (r, g, b));

        Sleep (time / steps);
        i++;
        }
    }

//---------------------------------------------------------------------------

void AppearText (int x, int y, const char* text, COLORREF from, COLORREF to,
                 int time, int steps)
    {
     int r0 = txExtractColor (from, TX_RED),   r1 = txExtractColor (to, TX_RED),
         g0 = txExtractColor (from, TX_GREEN), g1 = txExtractColor (to, TX_GREEN),
         b0 = txExtractColor (from, TX_BLUE),  b1 = txExtractColor (to, TX_BLUE);

    for (int i = 0; i <= steps; i++)
        {
        int r = r0 + (r1 - r0) * i/steps,
            g = g0 + (g1 - g0) * i/steps,
            b = b0 + (b1 - b0) * i/steps;

        txSetColor (RGB (r, g, b));
        txTextOut (x, y, text);

        Sleep (time / steps);
        }
    }

//---------------------------------------------------------------------------

void DrawFlag (int x, int y, int sizeX, int sizeY, COLORREF color, COLORREF bkColor)
    {
    txSetColor     (color);
    txSetFillColor (bkColor);

    txLine (x, y, x, y - sizeY);
    txRectangle (x, y - sizeY/2, x + sizeX, y - sizeY);

    txSelectFont ("Times New Roman", sizeX/2 + 1);
    txTextOut (x + sizeX/2, y - sizeY*7/8, "C++");
    }

//---------------------------------------------------------------------------

void DrawHello (int x, int y, const char* text, int size, COLORREF color)
    {
    txSetColor (color);
```

```
    txSelectFont ("Times New Roman", size);
    txSetTextAlign (TA_CENTER);

    txTextOut (x, y, text);
    }

//---------------------------------------------------------------------------

void DrawFrame (int sizeX, int sizeY, int size, COLORREF color)
    {
    txSetColor (color);
    txSetFillColor (TX_TRANSPARENT);

    txRectangle (size, size, sizeX - size, sizeY - size);
    }
```

Ниже более сложный пример, который тоже надо разобрать. Он тоже входит в состав примеров TXLib (Пример: Циклы (2)). Придется полазить по системе помощи TXLib, по Интернету и по учебникам в поисках понимания работы некоторых функций и конструкций языка.

```
#include <conio.h>
#include "TXLib.h"

//---------------------------------------------------------------------------

void DanceMan    (int x, int y, int sizeX, int sizeY, double jump,
                  COLORREF color, COLORREF bkColor, int delay);

void JumpMan     (int x, int y, int sizeX, int sizeY, double jump,
                  COLORREF color, COLORREF bkColor, int jumps, int delay);

void MoveMan     (int fromX, int fromY, int toX, int toY,
                  int sizeX, int sizeY, COLORREF color, COLORREF bkColor,
                  int time, int steps);

void DrawMan     (int x, int y, int sizeX, int sizeY, COLORREF color,
                  double hand = 0, double legs = 0, double head = 0, double twist = 0);

void AppearEarth (int x, int y, int sizeX, int sizeY, COLORREF from, COLORREF to,
                  int time, int steps);

void DrawEarth   (int x, int y, int sizeX, int sizeY, COLORREF color);

void AppearText  (int x, int y, const char* text, COLORREF from, COLORREF to,
                  int time, int steps);

void DrawHello   (int x, int y, const char* text, int size, COLORREF color);

void UnwindFlag  (int x, int y, int fromSizeX, int toSizeX, int sizeY,
                  COLORREF color, COLORREF bkColor, int time, int steps);

void DrawFlag    (int x, int y, int sizeX, int sizeY, COLORREF color, COLORREF bkColor);

void DrawFrame   (int sizeX, int sizeY, int size, COLORREF color);
```

```
int kbget();

//---------------------------------------------------------------------------

int main()
    {
    txCreateWindow (800, 600);
    txTextCursor (false);

    int sizeX   = txGetExtentX(), sizeY   = txGetExtentY();
    int centerX = (sizeX+1)/2,    centerY = (sizeY+1)/2;

    DrawFrame (sizeX, sizeY, 10, TX_WHITE);

    txSelectFont ("Times New Roman", 60);
    txSetTextAlign (TA_CENTER);

    AppearText  (centerX, sizeY*4/5, "\"Hello, world!\\n\" :)",    TX_BLACK,  TX_LIGHTGREEN, 5000,
100);
    AppearEarth (centerX, centerY, sizeX/2, sizeY/2,               TX_BLACK,  TX_LIGHTCYAN,  5000,
100);

    MoveMan (20, centerY - sizeY/4, centerX - sizeX/50, centerY - sizeY/4, sizeX/40, sizeY/15,
                                                                   TX_YELLOW, TX_BLACK,      3000,
100);
    txSleep (150);
    JumpMan (centerX - sizeX/50, centerY - sizeY/4, sizeX/40, sizeY/15, 0.25,
                                                                   TX_YELLOW, TX_BLACK,      10,
100);
    txPlaySound ("C:\\Windows\\Media\\tada.wav");

    UnwindFlag (centerX, centerY - sizeY/4, 0, sizeX/20, sizeY/10, TX_YELLOW, TX_BLACK,      500,
100);

    DanceMan   (centerX - sizeX/50, centerY - sizeY/4, sizeX/40, sizeY/15, 0.25,
                                                                   TX_YELLOW, TX_BLACK,      200);
    return 0;
    }

//---------------------------------------------------------------------------

void JumpMan (int x, int y, int sizeX, int sizeY, double jump,
              COLORREF color, COLORREF bkColor, int jumps, int delay)
    {
    DrawMan (x, y, sizeX, sizeY, TX_BLACK, 0, 0, 0, 0);

    txBegin();

    for (int i = 0; i < jumps && !_kbhit(); i++)
        {
        DrawMan (x, y - (int) (i%2 * jump*10), sizeX, sizeY, color,
                 (i%2 - 0.5) * jump/3, 0, (i%2 - 0.5) * -jump/3, 0);

        txSleep (delay);

        DrawMan (x, y - (int) (i%2 * jump*10), sizeX, sizeY, bkColor,
```

```
                (i%2 - 0.5) * jump/3, 0, (i%2 - 0.5) * -jump/3, 0);
        }

    kbget();

    DrawMan (x, y - (int) (jumps%2 * jump*10), sizeX, sizeY, color,
            (jumps%2 - 0.5) * jump/3, 0, (jumps%2 - 0.5) * -jump/3, 0);

    txEnd();
    }

//---------------------------------------------------------------------------

void DanceMan (int x, int y, int sizeX, int sizeY, double jump,
               COLORREF color, COLORREF bkColor, int delay)
    {
    DrawMan (x, y, sizeX, sizeY, TX_BLACK, 0, 0, 0, 0);

    txBegin();

    int i = 0;
    for (; !_kbhit(); i++)
        {
        DrawMan  (x, y - (int) (i%2 * jump*10), sizeX, sizeY, color,
                 (i%2 - 0.5) * jump/3, 0, (i%2 - 0.5) * -jump/3, (i%2 - 0.5) * jump);

        txSleep (delay);

        DrawMan  (x, y - (int) (i%2 * jump*10), sizeX, sizeY, bkColor,
                 (i%2 - 0.5) * jump/3, 0, (i%2 - 0.5) * -jump/3, (i%2 - 0.5) * jump);
        }

    kbget();

    DrawMan (x, y - (int) (i%2 * jump*10), sizeX, sizeY, color,
            (i%2 - 0.5) * jump/3, 0, (i%2 - 0.5) * -jump/3, (i%2 - 0.5) * jump);

    txEnd();
    }

//---------------------------------------------------------------------------

void MoveMan (int fromX, int fromY, int toX, int toY,
              int sizeX, int sizeY, COLORREF color, COLORREF bkColor,
              int time, int steps)
    {
    txBegin();

    for (int i = 0; i <= steps && !_kbhit(); i++)
        {
        int x = fromX + (toX - fromX) * i/steps,
            y = fromY + (toY - fromY) * i/steps;

        DrawMan (x, y - i%6, sizeX, sizeY, color,   i%3 * 0.02, i%3 * -0.1, i%3 * 0.1, 0);

        txSleep (time / steps);
```

```
        DrawMan (x, y - i%6, sizeX, sizeY, bkColor, i%3 * 0.02, i%3 * -0.1, i%3 * 0.1, 0);
        }

    kbget();

    DrawMan (toX, toY, sizeX, sizeY, color, 0, 0, 0);

    txEnd();
    }

//---------------------------------------------------------------------------

void UnwindFlag (int x, int y, int fromSizeX, int toSizeX, int sizeY,
                 COLORREF color, COLORREF bkColor, int time, int steps)
    {
    txBegin();

    for (int i = 0; i <= steps && !_kbhit(); i++)
        {
        int sizeX = fromSizeX + (toSizeX - fromSizeX) * i/steps;

        DrawFlag (x, y, sizeX, sizeY, color,   bkColor);

        txSleep  (time / steps);

        DrawFlag (x, y, sizeX, sizeY, bkColor, bkColor);
        }

    kbget();

    DrawFlag (x, y, toSizeX, sizeY, color, bkColor);

    txEnd();
    }

//---------------------------------------------------------------------------

void DrawMan (int x, int y, int sizeX, int sizeY, COLORREF color,
              double hand, double legs, double head, double twist)
    {
    txSetColor     (color);
    txSetFillColor (color);

    txLine (x + twist*sizeX, y - 0.35*sizeY, x, y - 0.7*sizeY);

    txLine (x + twist*sizeX, y - 0.35*sizeY, x - (0.5 + legs) * sizeX, y);
    txLine (x + twist*sizeX, y - 0.35*sizeY, x + (0.5 + legs) * sizeX, y);

    txLine (x, y - 0.65*sizeY, x - sizeX/2,   y - 0.4*sizeY);
    txLine (x, y - 0.65*sizeY, x + sizeX/1.2, y - (0.7 + hand) * sizeY);

    txCircle (x, y - sizeY + (0.3 + head) * sizeX, 0.3*sizeX);
    }

//---------------------------------------------------------------------------
```

```
// (Отступ в этой функции уменьшен, чтобы строчки с циклами поместились в ширину страницы,
//  и чтобы не разрывать соответствия между соответствием букв X и Y в именах по вертикали)

void DrawEarth (int x, int y, int sizeX, int sizeY, COLORREF color)
  {
  txSetColor (color);
  for (int rx = sizeX/2; rx >= 0; rx -= sizeX/9) txEllipse (x-rx,      y-sizeY/2, x+rx,
y+sizeY/2);
  for (int ry = sizeY/2; ry >= 0; ry -= sizeY/6) txEllipse (x-sizeX/2, y-ry,      x+sizeX/2, y+ry);
  txLine (x - sizeX/2, y, x + sizeX/2, y);
  }

//---------------------------------------------------------------------------

void AppearEarth (int x, int y, int sizeX, int sizeY, COLORREF from, COLORREF to,
                  int time, int steps)
    {
    int r0 = txExtractColor (from, TX_RED),   r1 = txExtractColor (to, TX_RED),
        g0 = txExtractColor (from, TX_GREEN), g1 = txExtractColor (to, TX_GREEN),
        b0 = txExtractColor (from, TX_BLUE),  b1 = txExtractColor (to, TX_BLUE);

    for (int i = 0; i <= steps && !_kbhit(); i++)
        {
        int r = r0 + (r1 - r0) * i/steps,
            g = g0 + (g1 - g0) * i/steps,
            b = b0 + (b1 - b0) * i/steps;

        DrawEarth (x, y, sizeX, sizeY, RGB (r, g, b));

        Sleep (time / steps);
        }

    kbget();

    DrawEarth (x, y, sizeX, sizeY, to);
    }

//---------------------------------------------------------------------------

void AppearText (int x, int y, const char* text, COLORREF from, COLORREF to,
                 int time, int steps)
    {
    int r0 = txExtractColor (from, TX_RED),   r1 = txExtractColor (to, TX_RED),
        g0 = txExtractColor (from, TX_GREEN), g1 = txExtractColor (to, TX_GREEN),
        b0 = txExtractColor (from, TX_BLUE),  b1 = txExtractColor (to, TX_BLUE);

    for (int i = 0; i <= steps && !_kbhit(); i++)
        {
        int r = r0 + (r1 - r0) * i/steps,
            g = g0 + (g1 - g0) * i/steps,
            b = b0 + (b1 - b0) * i/steps;

        txSetColor (RGB (r, g, b));
        txTextOut (x, y, text);

        Sleep (time / steps);
```

```
        }

    kbget();

    txSetColor (to);
    txTextOut (x, y, text);
    }

//---------------------------------------------------------------------------

void DrawFlag (int x, int y, int sizeX, int sizeY, COLORREF color, COLORREF bkColor)
    {
    txSetColor     (color);
    txSetFillColor (bkColor);

    txLine (x, y, x, y - sizeY);
    txRectangle (x, y - sizeY/2, x + sizeX, y - sizeY);

    txSelectFont ("Times New Roman", sizeX/2 + 1);
    txTextOut (x + sizeX/2, y - sizeY*7/8, "C++");
    }

//---------------------------------------------------------------------------

void DrawHello (int x, int y, const char* text, int size, COLORREF color)
    {
    txSetColor (color);

    txSelectFont ("Times New Roman", size);
    txSetTextAlign (TA_CENTER);

    txTextOut (x, y, text);
    }

//---------------------------------------------------------------------------

void DrawFrame (int sizeX, int sizeY, int size, COLORREF color)
    {
    txSetColor (color);
    txSetFillColor (TX_TRANSPARENT);

    txRectangle (size, size, sizeX - size, sizeY - size);
    }

//---------------------------------------------------------------------------

int kbget()
    {
    int ch = 0;
    while (_kbhit()) ch = _getch();
    return ch;
    }
```

1.4.3. Еще один пример

Еще один пример кода с говорящим названием MinecraftStory.cpp, посвященного одной известной игре, можно посмотреть в примерах к TXLib (папка TX\Examples\Demo). Автор кода – семиклассник, и на момент обучения не самый выдающийся программист в классе. Потом, конечно, у него все отлично сложилось с программированием.

Вот как приблизительно выглядит одна из функций, реализующих сцену движения объекта:

```
void RunToTree()
    {
    int t = 0;
    while (t <= 100)
        {
        txClear();
        Scenery3();

        Cloud (340 - t * 8, 70, TX_WHITE, TX_BLACK, 1.2, 0, 30, 15);
        Cloud (750 - t * 10, 160, RGB (247, 245, 245), TX_BLACK, 1, 20, 0, 0);

        Man (810 - 3 * t, 570 - t % 10, 1, -1, RGB (33, 29, 146), RGB (30, 148, 178), 1);
        Sun (1150 + 1.4 * t, 90 + 0.1 * t, RGB (247, 225, 82), 1, 0);

        txSleep (100);
        t++;
        }
    }
```

Эта функция очень простая. Здесь всего один "скачущий" параметр, заданный формулой 570 - t%10, он управляет положением человечка по координате Y. Было бы хорошо, конечно, оживить сцену, заставив двигаться и дергаться другие части объектов. Но в данной работе автор остановился на этом, позволив вам пойти дальше него и сделать ваши мультфильмы гораздо лучше и интереснее.

В конце концов, мы здесь не "для галочки" работаем. Сложнее сцены – больше кода, больше опыт. Только культуру соблюдайте, а то вместо прогресса будет деградация.

1.6. Библиотеки. Документация

Итак, функции движения и рисования написаны, но можно заметить, что файл с программой стал уже слишком большой.

Это похоже на то, как получилось с первой программой: пока мы не разделили функцию main на части (другие функции), она была длинной и непонятной. Применив принцип "разделяй и властвуй", мы легко сделали код ясным, модульным и более удобным для программиста. Логично появляется вопрос, можно ли что-то подобное сделать с длинным файлом, ставшим слишком сложным. Этот вопрос можно задать знакомому-профессионалу как раз в такой форме: "я могу разделить длинную функцию на другие функции, как я могу сделать то же с длинным файлом", или "как разделить длинный файл в C++ на другие файлы"?

Ответов на этот вопрос два, один – так называемая раздельная компиляция, но в ней поначалу легко запутаться. Второй, подходящий для небольших проектов – очень простой, и мы фактически уже им давно пользуемся с первого занятия: подключение библиотек директивой

#include. Эта директива заставляет компилятор Си просмотреть сначала файл, указанный после #include, а затем обработать нашу программу. Такой директивой мы подключали библиотечный файл TXLib.h. Действуя этим способом, мы можем вынести прототипы и определения функций во вспомогательный файл-библиотеку с логичным названием (например, DrawLib.cpp) и подключить его к основной программе.

Что после этого получится:

Файл HeroesLib.cpp:

```
#include "TXLib.h"

//-----------------------------------------------------------------------------------------

void DrawMan (int x, int y, double sizeX, double sizeY);
...

//-----------------------------------------------------------------------------------------

void DrawMan (int x, int y, double sizeX, double sizeY)
    {
    txSetFillColor (TX_NULL);

    txLine (x, y,             x - 50*sizeX, y +  50*sizeY);
    txLine (x, y,             x + 50*sizeX, y +  50*sizeY);
    txLine (x, y,             x,            y + 100*sizeY);
    txLine (x, y + 100*sizeY, x - 50*sizeX, y + 150*sizeY);
    txLine (x, y + 100*sizeY, x + 50*sizeX, y + 150*sizeY);

    txCircle (x, y - 50*sizeY, 50*sizeY);
    }

...
```

Файл Program.cpp:

```
#include "TXLib.h"

#include "HeroesLib.cpp"  // Этот файл тоже включает TXLib.h, но от повторного включения
                          // TXLib ничего страшного не произойдет.

//-----------------------------------------------------------------------------------------

void PoliceChase();
...

//-----------------------------------------------------------------------------------------

int main()
    {
    ...

    PoliceChase();
    ...

    }

//-----------------------------------------------------------------------------------------

void PoliceChase()
    {
```

```
    double t = 0;
    while (t <= 100)
        {
        txSetFillColor (TX_BLACK);
        txClear();

        DrawMan        (400 +   3*t, 250 + 0.6*t);
        DrawMan        ( 50 +   6*t, 270 + 0.6*t);
        DrawWoman      (250 +   4*t, 400 - 0.3*t);
        DrawHelicopter (600 + 0.5*t,   0 + 1.5*t);

        t++;

        txSleep (100);
        }
    }

...
```

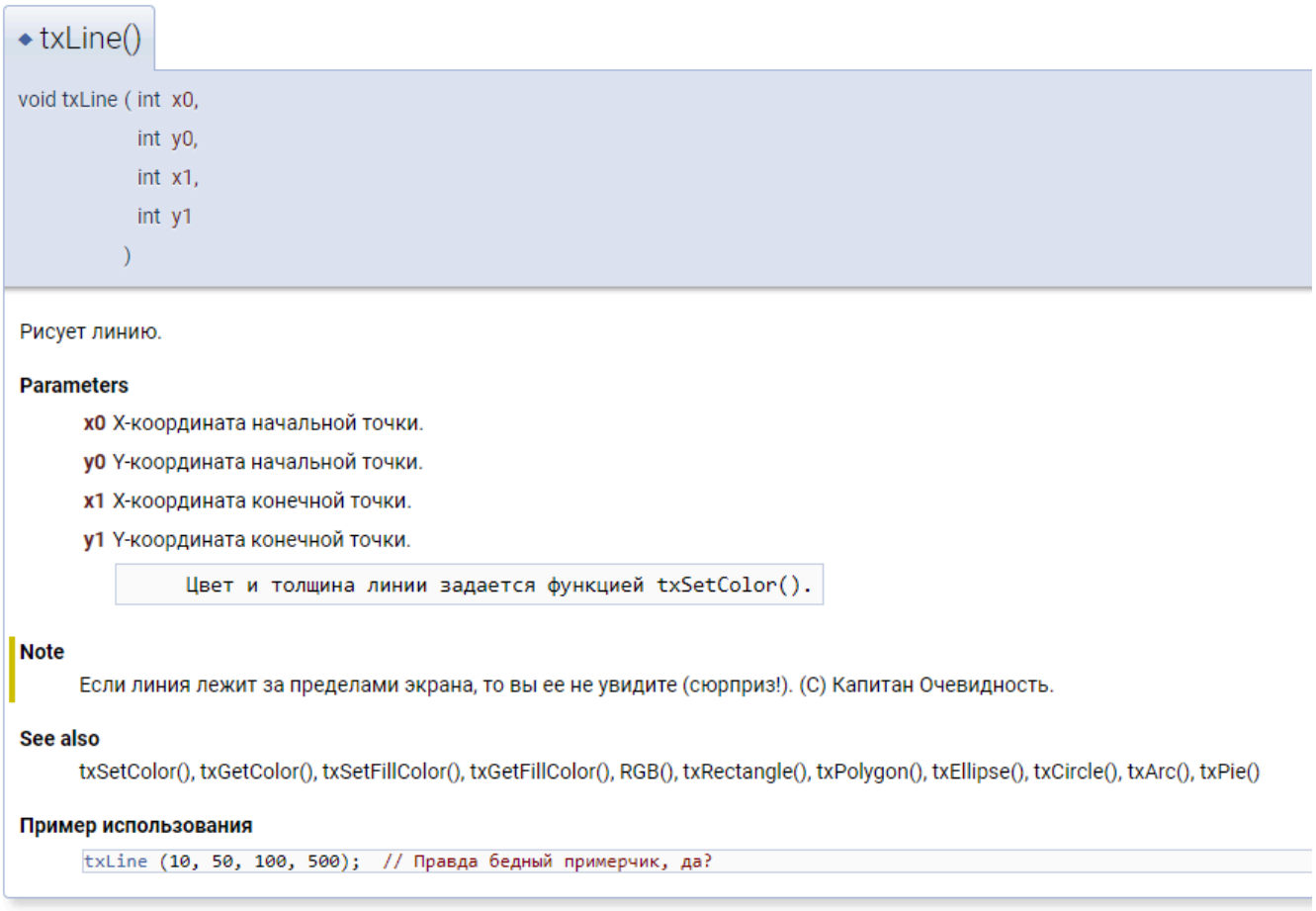
Какие функции включать в библиотеку, а какие оставлять в основной программе? Лучше выносить в библиотеку то, что может пригодиться в дальнейшем: рисование героев, фонов, типичные сцены и движения. Не стоит выносить то, что специфично для конкретного сюжета мультфильма и вряд ли пригодиться в подобных проектах. Такой библиотечный файл будет очень полезен для дальнейшего развития нашей работы. Например, мы хотим создать вторую серию мультфильма или выпустить новый уровень игры с теми же героями. Тогда их не придется писать заново или копировать из файла в файл, увеличивая путаницу и объем кода.

Кроме того, библиотеками можно меняться с другими разработчиками. Но для этого их надо сделать более понятными, так как другие люди не всегда быстро поймут ход ваших мыслей и точный смысл названий ваших функций, констант и параметров. Для этого есть простой (и, к счастью, правильный) способ – написать так называемые документирующие комментарии и на их основе автоматически сгенерировать документацию. Для такой генерации есть утилиты, и с помощью одной из таких утилит – Doxygen – создана система помощи TXLib, которой вы пользуетесь. Рассмотрим, например, документирующие комментарии для слегка упрощенной версии функции txLine:

```
//{-----------------------------------------------------------------------------------------
//! Рисует линию.
//!
//! @param   x0    X-координата начальной точки.
//! @param   y0    Y-координата начальной точки.
//! @param   x1    X-координата конечной  точки.
//! @param   y1    Y-координата конечной  точки.
//!
//!          Цвет и толщина линии задается функцией txSetColor().
//!
//! @note    Если линия лежит за пределами экрана, то вы ее не увидите (сюрприз!).
//!          (C) Капитан Очевидность.
//!
//! @see     txSetColor(), txGetColor(), txSetFillColor(), txGetFillColor(), RGB(),
//!          txRectangle(), txPolygon(), txEllipse(), txCircle(), txArc(), txPie()
//!
//! @par Пример использования
//! @code
//!          txLine (10, 50, 100, 500);  // Правда бедный примерчик, да?
//! @endcode
//}-----------------------------------------------------------------------------------------

void txLine (int x0, int y0, int x1, int y1);
```

Документирующие комментарии в формате Doxygen пишутся перед прототипом функции и начинаются с символов //!. Обычные комментарии, начинающиеся с символов //, выполняют роль разделителей, повышая структурность текста. Ключевые слова (теги) Doxygen начинаются с символа @ и задают описания параметров, создают разделы текста (@note), ссылки на другие функции (@see), заголовки параграфов (@par) и фрагменты кода (@code / @endcode). Ключевых слов у Doxygen довольно много, их можно прочесть в официальной документации к этому пакету (найдите ее), но начать лучше со статей на Хабре (поищите по запросу habr doxygen, их там много).

Фрагмент системы помощи, соответствующей этим документирующим комментариям, будет выглядеть так:



txLine()

```
void txLine ( int x0,
              int y0,
              int x1,
              int y1
            )
```

Рисует линию.

Parameters

x0 X-координата начальной точки.

y0 Y-координата начальной точки.

x1 X-координата конечной точки.

y1 Y-координата конечной точки.

```
Цвет и толщина линии задается функцией txSetColor().
```

Note

Если линия лежит за пределами экрана, то вы ее не увидите (сюрприз!). (C) Капитан Очевидность.

See also

txSetColor(), txGetColor(), txSetFillColor(), txGetFillColor(), RGB(), txRectangle(), txPolygon(), txEllipse(), txCircle(), txArc(), txPie()

Пример использования

```
txLine (10, 50, 100, 500);  // Правда бедный примерчик, да?
```

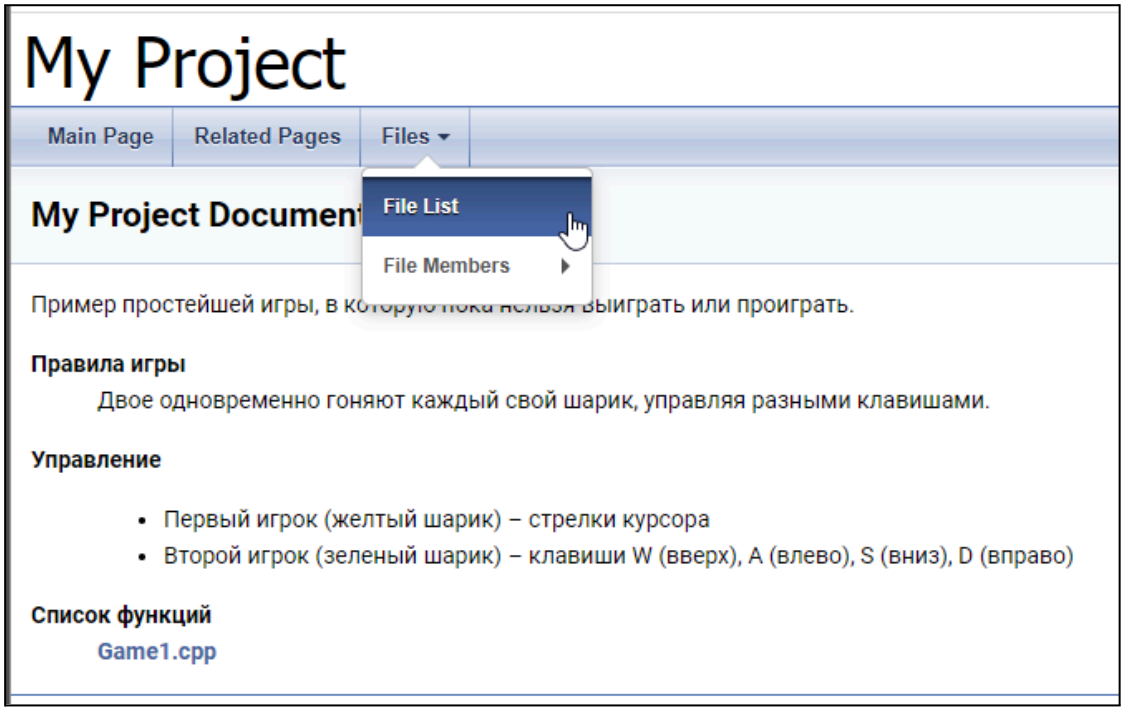
Попробуйте найти в исходном тексте документирующие комментарии других функций в файле TXLib.h (правда, этот файл большой) и сравнить с соответствующими фрагментами системы помощи TXLib.

Кроме того, чтобы Doxygen эффективно обработал ваш файл, нужно написать документацию на файл в целом. Разместите в начале файла, например:

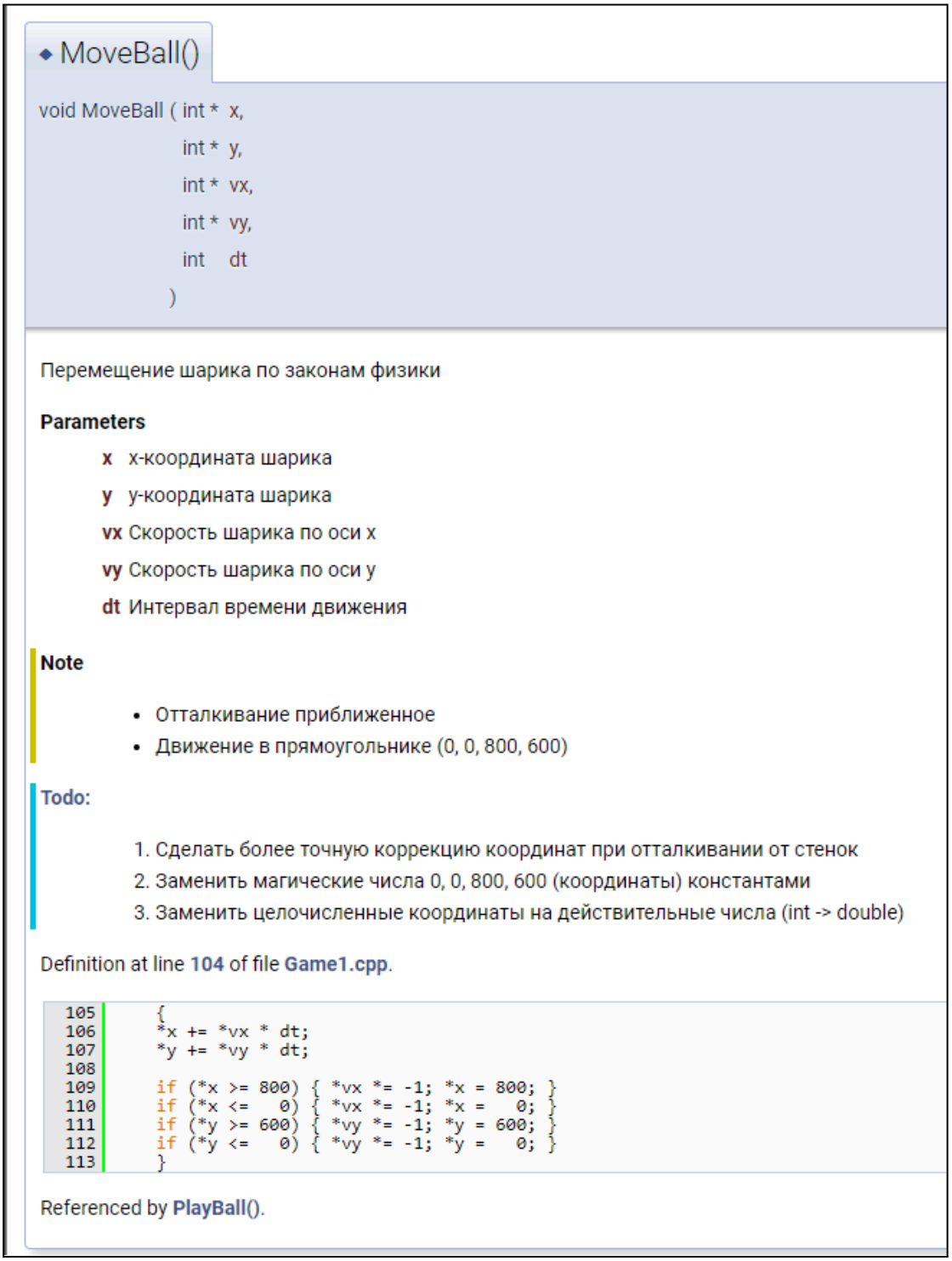
```
//=============================================================================================
//! @file TXLib.h
//!
//! Библиотека Тупого Художника (The Dumb Artist Library, TX Library, TXLib).
//!
//! @author  Ded (Ilya Dedinsky, http://txlib.ru) <mail@txlib.ru>
//! @version 1.0
//! @date    Date: 2005-09-05
//!
//! @mainpage
//!
```

```
//! TX Library - компактная библиотека двумерной графики для MS Windows на C++.
//!
//! - Философия TX Library - облегчить первые шаги в программировании и подтолкнуть к творчеству
//!   и самостоятельности.
//! - Посмотрите @ref Screenshots "Скриншоты!"
//! - Замените текст на свой. Вы же не автор TXLib (пока).
//!
//! @note {@link http://txlib.ru Официальный сайт библиотеки.}
//!
//! @section Screenshots Скриншоты
//!
//! @image html Example03.png "Example03.cpp: Простейшая программа"
//!
//! ...
//! (и так далее)
//! ...
//!
//=============================================================================================
```

Здесь @mainpage – тег, задающий текст на главной странице документации, @link – вставка гиперссылки, @section – создание раздела, @image – вставка картинки. Главная страница вашей документации, использующая эти теги, может выглядеть так:



Документация к функциям проекта, если в настройках вы включите все опции, указанные ниже, будет выглядеть так:



◆ MoveBall()

```
void MoveBall ( int * x,
                int * y,
                int * vx,
                int * vy,
                int   dt
              )
```

Перемещение шарика по законам физики

Parameters

- **x** x-координата шарика
- **y** y-координата шарика
- **vx** Скорость шарика по оси x
- **vy** Скорость шарика по оси y
- **dt** Интервал времени движения

Note

- Отталкивание приближенное
- Движение в прямоугольнике (0, 0, 800, 600)

Todo:

1. Сделать более точную коррекцию координат при отталкивании от стенок
2. Заменить магические числа 0, 0, 800, 600 (координаты) константами
3. Заменить целочисленные координаты на действительные числа (int -> double)

Definition at line **104** of file **Game1.cpp**.

```
105     {
106     *x += *vx * dt;
107     *y += *vy * dt;
108
109     if (*x >= 800) { *vx *= -1; *x = 800; }
110     if (*x <=   0) { *vx *= -1; *x =   0; }
111     if (*y >= 600) { *vy *= -1; *y = 600; }
112     if (*y <=   0) { *vy *= -1; *y =   0; }
113     }
```

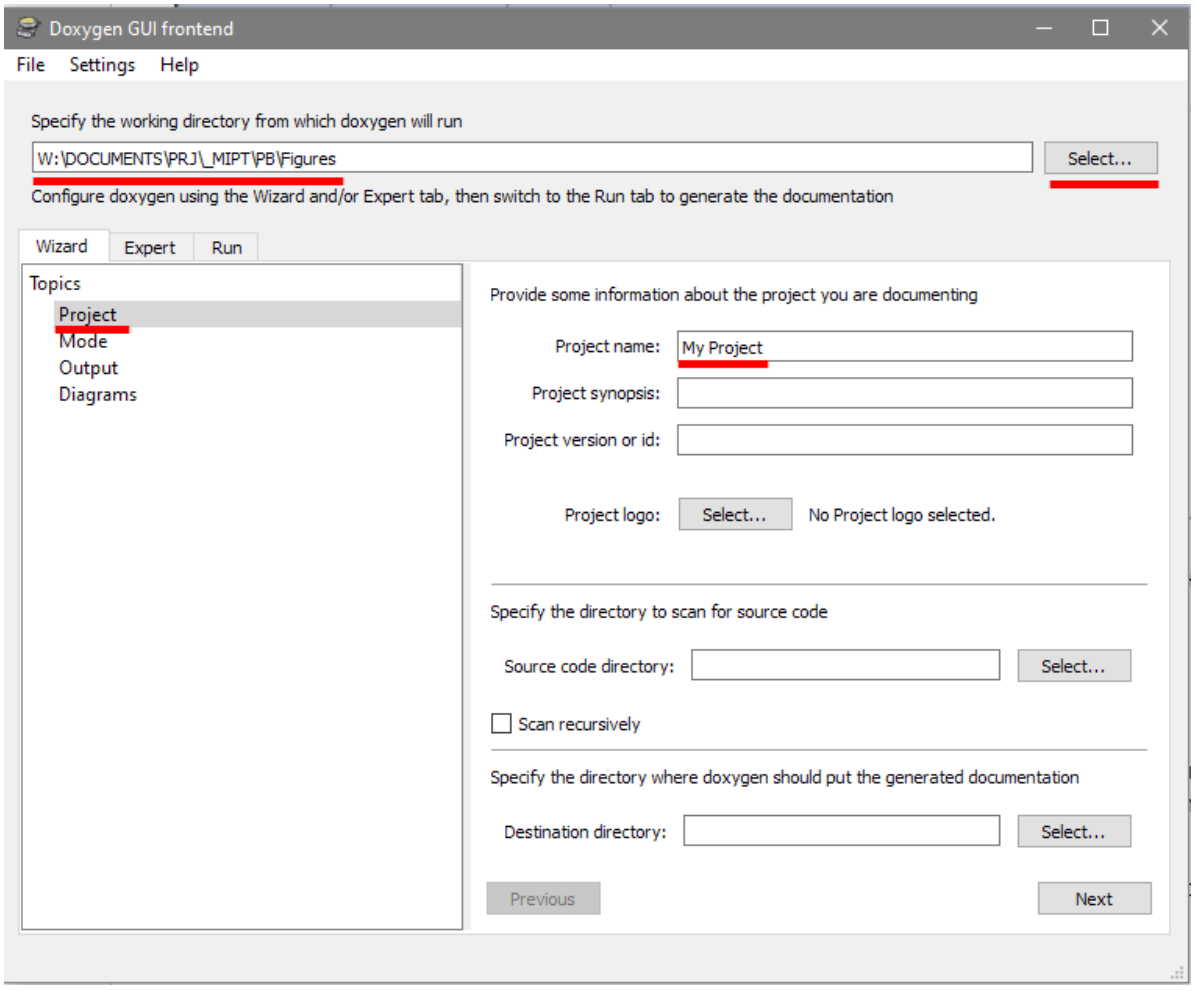
Referenced by **PlayBall()**.

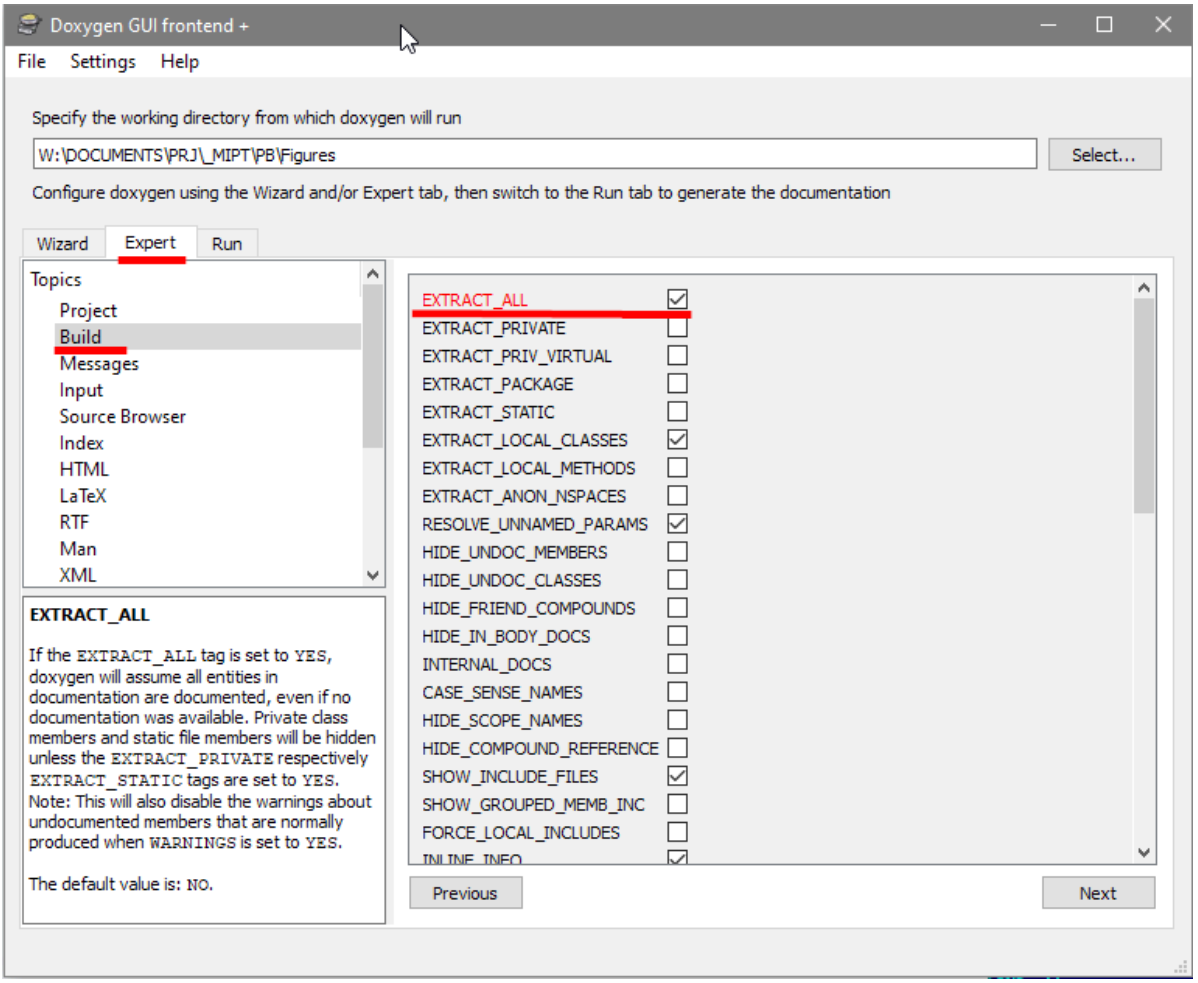
Разумеется, имя My Project и прочие названия по умолчанию надо заменить на осмысленные собственные имена.

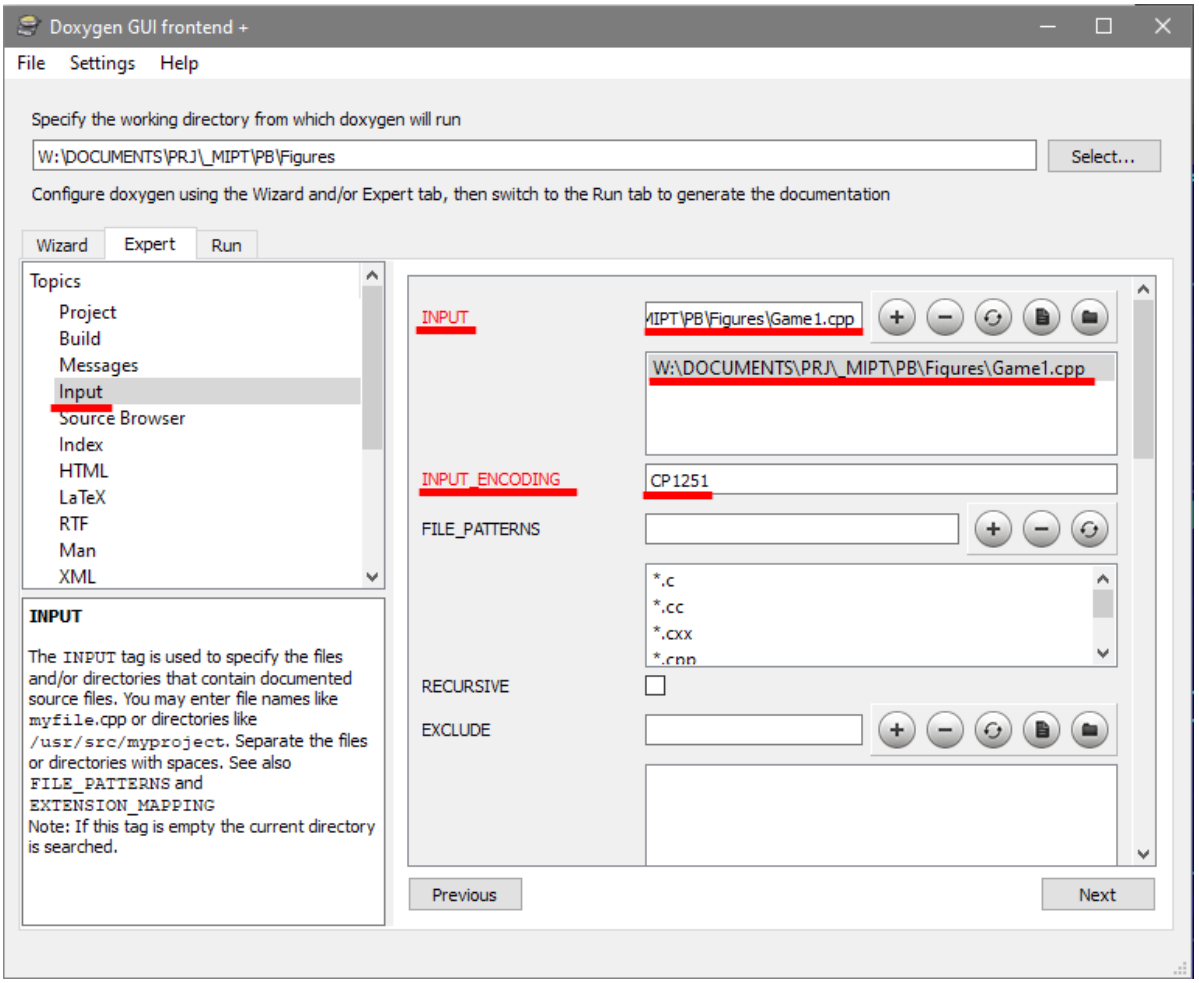
Написав документирующие комментарии, надо преобразовать их в систему помощи. Делается это так.

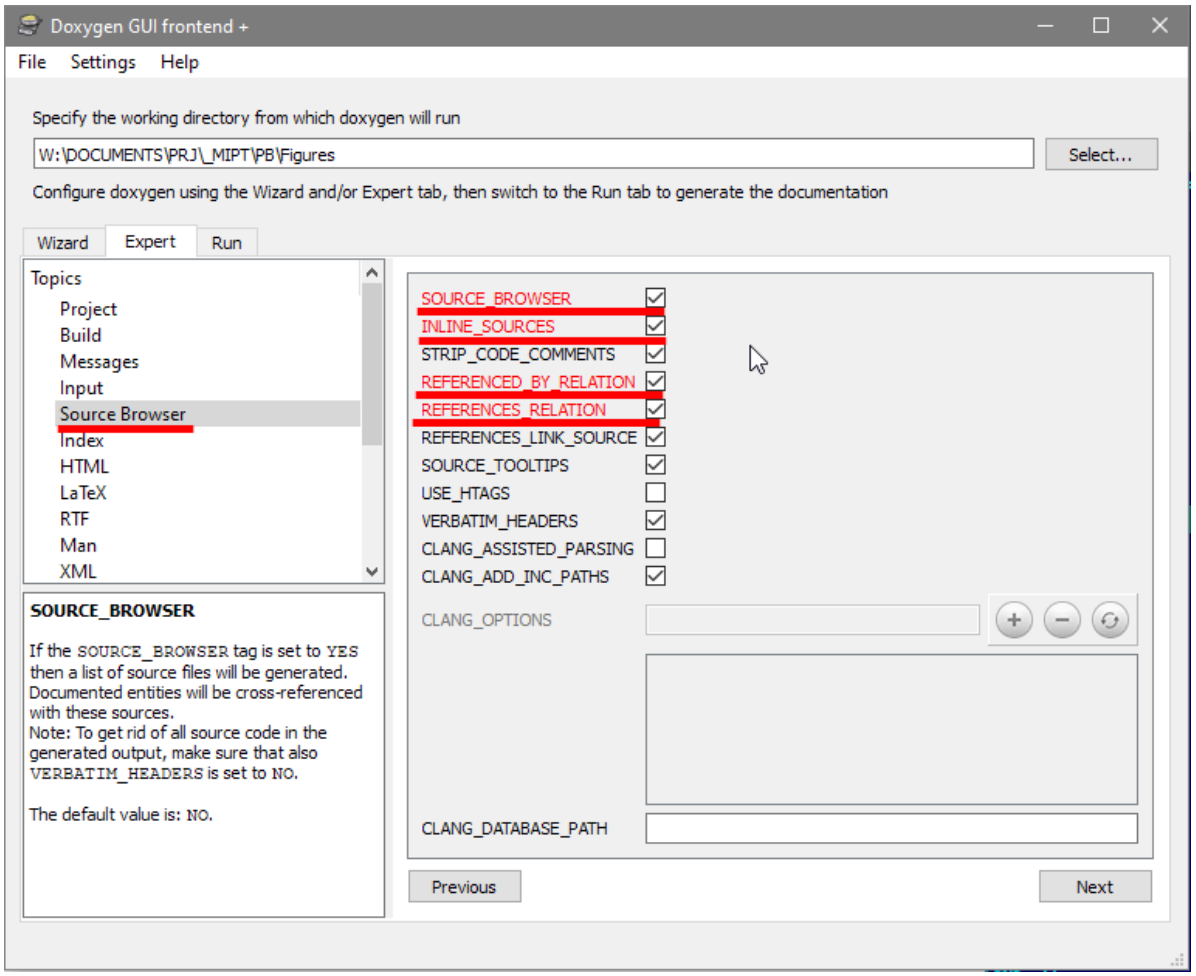
Сначала надо зайти на сайт doxygen.nl, найти там страницу загрузки (Downloads), скачать и установить систему Doxygen. Если вы используете сборку CodeBlocks "с правильными настройками" с сайта ded32.ru, то в ней уже есть установленная система doxygen.

После этого надо запустить программу DoxyWizard, заполнить необходимые поля и сохранить файл настроек Doxygen в папку с вашим проектом. Далее рекомендуется включить опции, показанные ниже на рисунках.

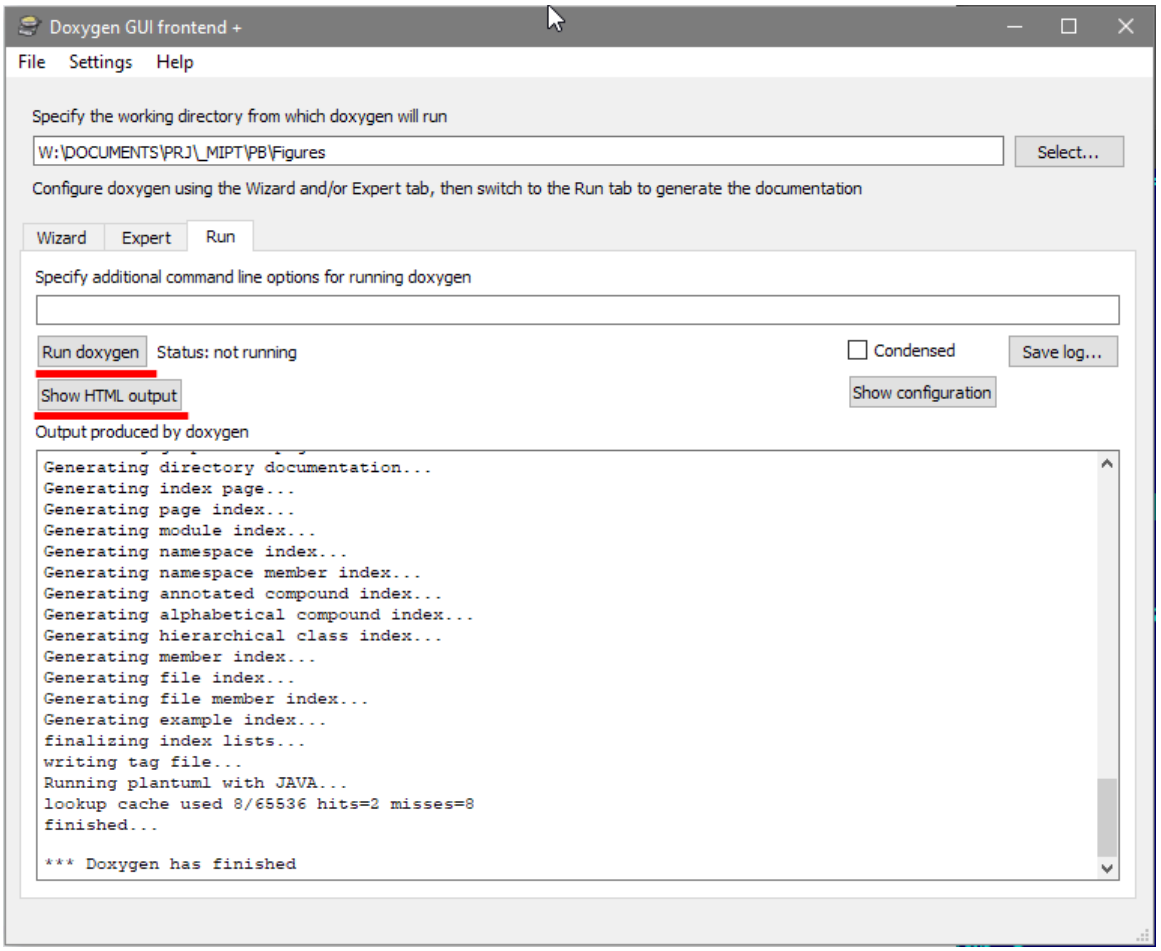

Doxygen GUI frontend
File Settings Help
Specify the working directory from which doxygen will run
W:\DOCUMENTS\PRJ_MIPT\PB\Figures
Select...
Configure doxygen using the Wizard and/or Expert tab, then switch to the Run tab to generate the documentation
Wizard Expert Run
Topics
Project
Mode
Output
Diagrams
Provide some information about the project you are documenting
Project name: My Project
Project synopsis:
Project version or id:
Project logo: Select... No Project logo selected.
Specify the directory to scan for source code
Source code directory: Select...
Scan recursively
Specify the directory where doxygen should put the generated documentation
Destination directory: Select...
Previous Next


Doxygen GUI frontend +
File Settings Help
Specify the working directory from which doxygen will run
W:\DOCUMENTS\PRJ_MIPT\PB\Figures
Select...
Configure doxygen using the Wizard and/or Expert tab, then switch to the Run tab to generate the documentation
Wizard Expert Run
Topics
Project
Build
Messages
Input
Source Browser
Index
HTML
LaTeX
RTF
Man
XML
EXTRACT_ALL
If the EXTRACT_ALL tag is set to YES, doxygen will assume all entities in documentation are documented, even if no documentation was available. Private class members and static file members will be hidden unless the EXTRACT_PRIVATE respectively EXTRACT_STATIC tags are set to YES. Note: This will also disable the warnings about undocumented members that are normally produced when WARNINGS is set to YES.
The default value is: NO.
EXTRACT_ALL
EXTRACT_PRIVATE
EXTRACT_PRIV_VIRTUAL
EXTRACT_PACKAGE
EXTRACT_STATIC
EXTRACT_LOCAL_CLASSES
EXTRACT_LOCAL_METHODS
EXTRACT_ANON_NSPACES
RESOLVE_UNNAMED_PARAMS
HIDE_UNDOC_MEMBERS
HIDE_UNDOC_CLASSES
HIDE_FRIEND_COMPOUNDS
HIDE_IN_BODY_DOCS
INTERNAL_DOCS
CASE_SENSE_NAMES
HIDE_SCOPE_NAMES
HIDE_COMPOUND_REFERENCE
SHOW_INCLUDE_FILES
SHOW_GROUPED_MEMB_INC
FORCE_LOCAL_INCLUDES
Previous Next


Doxygen GUI frontend +
File Settings Help
Specify the working directory from which doxygen will run
W:\DOCUMENTS\PRJ_MIPT\PB\Figures
Select...
Configure doxygen using the Wizard and/or Expert tab, then switch to the Run tab to generate the documentation
Wizard Expert Run
Topics
Project
Build
Messages
Input
Source Browser
Index
HTML
LaTeX
RTF
Man
XML
INPUT
The INPUT tag is used to specify the files and/or directories that contain documented source files. You may enter file names like myfile.cpp or directories like /usr/src/myproject. Separate the files or directories with spaces. See also FILE_PATTERNS and EXTENSION_MAPPING
Note: If this tag is empty the current directory is searched.
INPUT
MIPT\PB\Figures\Game1.cpp
W:\DOCUMENTS\PRJ_MIPT\PB\Figures\Game1.cpp
INPUT_ENCODING
CP1251
FILE_PATTERNS
*.c
*.cc
*.cxx
*.cpp
RECURSIVE
EXCLUDE
Previous
Next


Doxygen GUI frontend +
File Settings Help
Specify the working directory from which doxygen will run
W:\DOCUMENTS\PRJ_MIPT\PB\Figures
Select...
Configure doxygen using the Wizard and/or Expert tab, then switch to the Run tab to generate the documentation
Wizard Expert Run
Topics
Project
Build
Messages
Input
Source Browser
Index
HTML
LaTeX
RTF
Man
XML
SOURCE_BROWSER
If the SOURCE_BROWSER tag is set to YES then a list of source files will be generated. Documented entities will be cross-referenced with these sources.
Note: To get rid of all source code in the generated output, make sure that also VERBATIM_HEADERS is set to NO.
The default value is: NO.
SOURCE_BROWSER
INLINE_SOURCES
STRIP_CODE_COMMENTS
REFERENCED_BY_RELATION
REFERENCES_RELATION
REFERENCES_LINK_SOURCE
SOURCE_TOOLTIPS
USE_HTAGS
VERBATIM_HEADERS
CLANG_ASSISTED_PARSING
CLANG_ADD_INC_PATHS
CLANG_OPTIONS
CLANG_DATABASE_PATH
Previous
Next

После этого перейти во вкладку Run и нажать на кнопку “Run Doxygen”. После этого – на кнопку “Show HTML output”. В открывшемся окне браузера вы увидите нечто подобное скриншоту с документацией “My Project” выше.



Передавая другим свою библиотеку, надо документировать все функции, параметры, константы и другие конструкции. Это дело чести будущего профессионального программиста.

1.7. Примеры

К сожалению, документации недостаточно. Люди так устроены (и разработчики здесь не исключение), что они привыкли понимать все по примерам. Поэтому без законченных примеров использования библиотеки никто даже не откроет документацию.

Примеров использования лучше делать не меньше трех. Первый пример пусть будет самый элементарный, он продемонстрирует простоту использования. Например, если библиотека посвящена юбилею первого пилотируемого полета в космосе, то с ее помощью можно легко нарисовать космонавта, ракету и Землю. Без этого – сами понимаете, что будет: работу с библиотекой сочтут слишком сложной. Второй пример будет посложнее, лучше его дать как развитие первого, чтобы показать, что в библиотеке есть много возможностей. Например, можно нарисовать 42 любопытных инопланетянина с Марса, заинтересованно следящих за развитием космоса странной расой всего лишь двуруких, двуногих и удручающе двуглазых (и откуда вообще у этих землян эта любовь к числу 2?) существ. Также есть удобные функции для движения толпы инопланетян с Марса на Землю (на выпускной вечер 11-классников) и обратно (когда их туда не пустили). Третьим примером может быть полный исходный код вашего мультфильма – ведь он тоже сделан с использованием вашей библиотеки.

Положите ваши примеры в отдельную папку Examples, чтобы их легко нашли и включите файлы примеров в документацию. Назовите файлы так, чтобы было понятно, где простой пример, а где сложнее.

1.8. Ревью кода библиотеки

Редко когда хорошая библиотека или программа получается сразу. Точнее – вообще никакая сразу не получается. Приходится искать и исправлять ошибки, многое рефакторить. Разработка – непростой процесс, и не всегда в ней все очевидно. В том числе часто бывает такая ситуация, что автору все понятно и логично, а пользователям, коллегам, другим программистам – нет. Чтобы узнать мнение коллег о программе или библиотеке, проводят письменное ревью кода. В простейшем смысле это что-то вроде рецензии, но очень структурной и четкой:

1. Обязательно начните с указания, как называется библиотека, кто автор, какая версия, дата выпуска – по этой информации автор не спутает ваше ревью с каким-то другим или не выкинет его, думая, что оно относится к устаревшей версии;
2. Информации о ревьювере, как с ним связаться – иначе автор не сможет сказать даже "спасибо";
3. Какие разделы, файлы и функции она содержит (если их немного) – тут автор поймет, что вы вообще открывали и внимательно смотрели текст библиотеки;
4. Какие положительные моменты встретились в библиотеке – без этого автор обидится на критику и не будет читать ревью;
5. Недостатки кода в виде предложений по исправлению – иначе автор скажет, что критиковать все горазды, и предложит автору ревью исправить все самому;
6. Очень мало что еще.

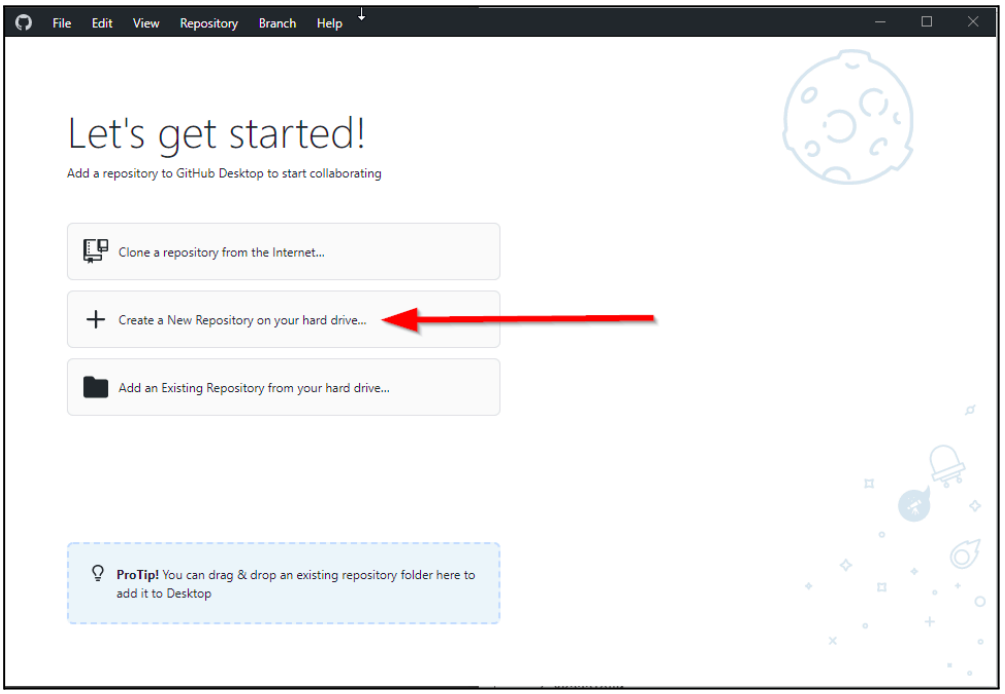
Ревью кода постоянно используется в профессиональной практике. Например, во многих компаниях код, не прошедший ревью, нельзя использовать в работе. Часто требуется два ревью – одно внутреннее (от ближайшего коллеги из той же команды, где разработчик) и одно внешнее (от разработчика из другой команды). В особо ответственных случаях существуют ревью на ревью. Все это происходит, чтобы разработчики продвигались к цели в условиях командной работы, а не постоянно ругались и рефакторили код по кругу, не достигая результата.

1.9. Версии

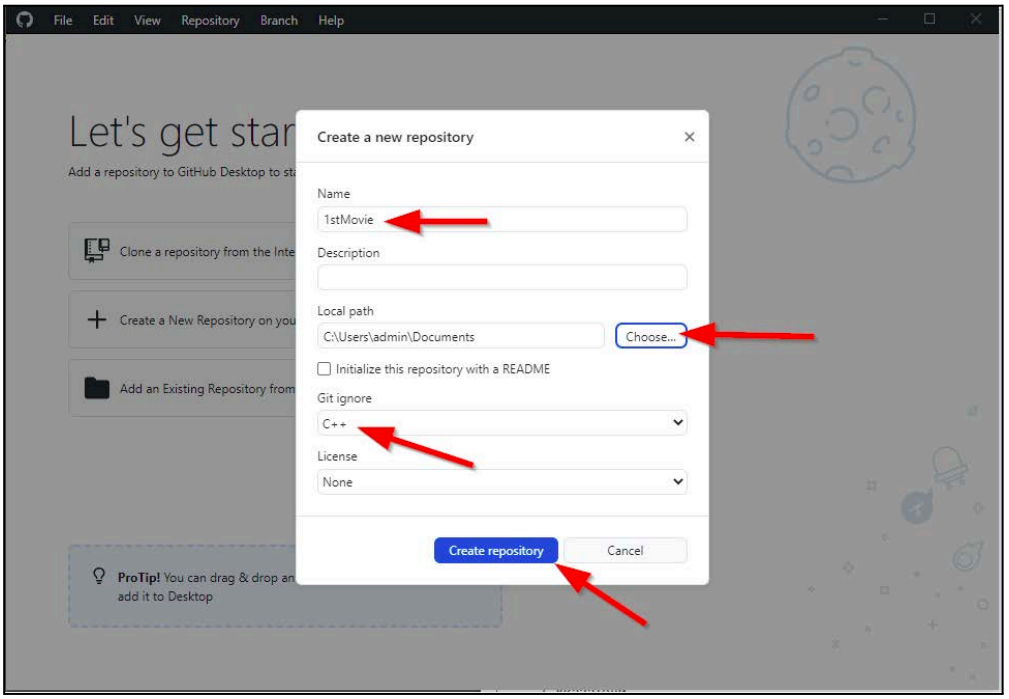
После ревью следует рефакторинг с учетом идей и пожеланий ревьюера (автор библиотеки не обязан соглашаться со всеми замечаниями). Получается новая версия библиотеки, удобнее и лучше. Выпуская новую версию, нужно не забыть обновить ее дату и номер в документации, всех аннотациях, названиях архивов и т.п., иначе сразу возникнет путаница "что где". Долгое время для каждой версии делался архив и складывался в папку с другими архивами предыдущих версий. Но так получалось неудобно: в большом количестве архивов все равно была путаница, неудобно было отслеживать изменения в версиях, следить, как нужный фрагмент кода менялся от версии к версии. Поэтому для решения этих задач придумали системы контроля версий. Эти системы контролируют файлы в проекте, который называется репозиторием, чтобы ни один файл и ни одно изменение файла не пропало и не перепуталось. Репозиторий – это папка на диске с базой данных, хранящей все версии всех добавленных туда файлов, причем все версии и все изменения в них можно легко просматривать, а при необходимости и восстанавливать обратно, если вы случайно что-то сломали в рабочем коде. Вообще, сложно представить, как программисты работали до систем контроля версий.

Систем контроля версий довольно много. Одна из самых известных таких систем - Git, созданная автором ядра Linux – Линусом Торвальдсом, как раз для разработки этого ядра. Это довольно сложная и, поначалу может показаться, запутанная система, но, к счастью, для нее есть программы-надстройки, облегчающие ее использование. Одна из таких надстроек - программа Github Desktop, работающая с сайтом Github.com. Скачать и установить программу Github Desktop можно с сайта desktop.github.com.

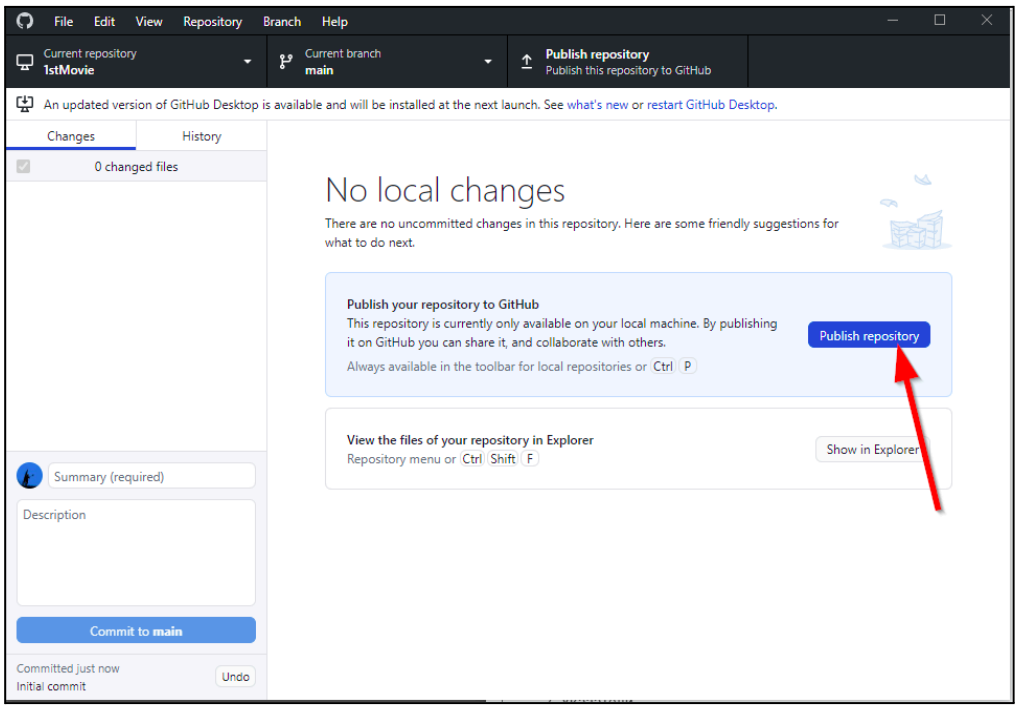
После установки надо зарегистрироваться на сайте github.com (нужно будет подтвердить адрес электронной почты) и ввести свой логин и пароль в Github Desktop. После этого можно создать репозиторий на своем компьютере (это называется локальный репозиторий), по сути создается обычная папка на вашем диске:



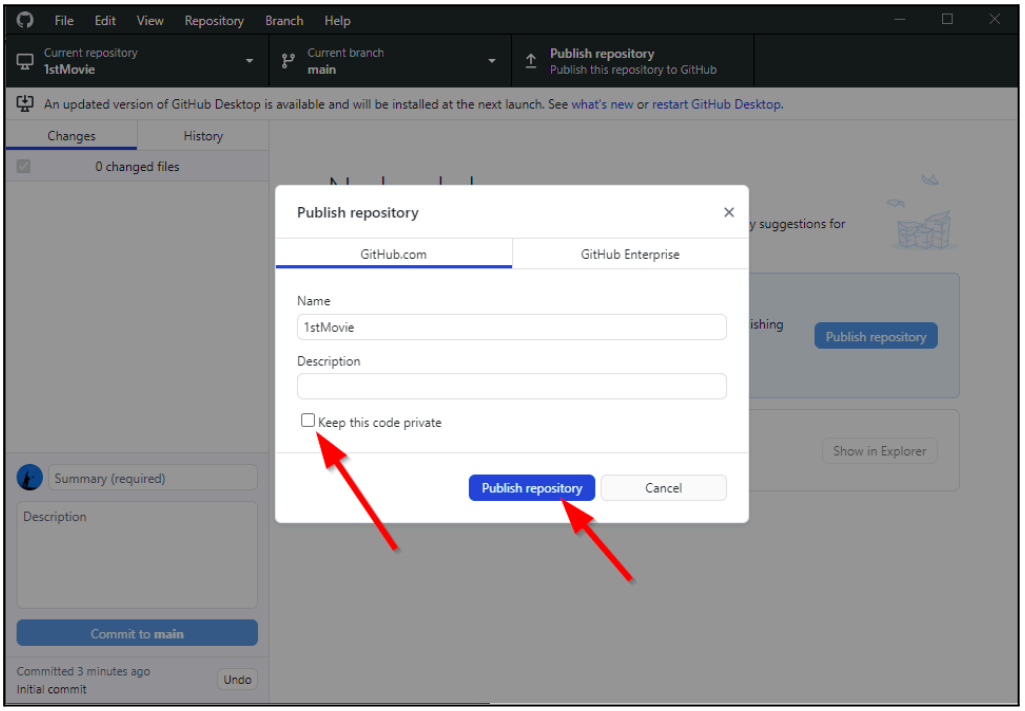
При создании вы указываете название репозитория (латинскими буквами и цифрами без пробелов, можно с дефисами), выбираете путь, где будет создана папка репозитория и тип проекта (C++):



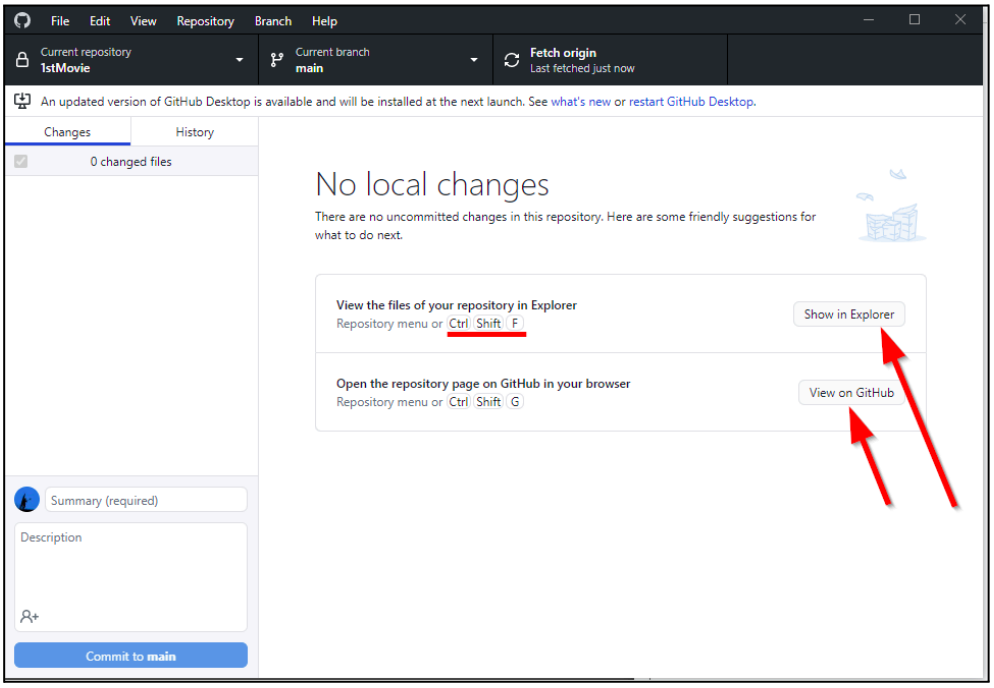
После создания репозиторий надо опубликовать:



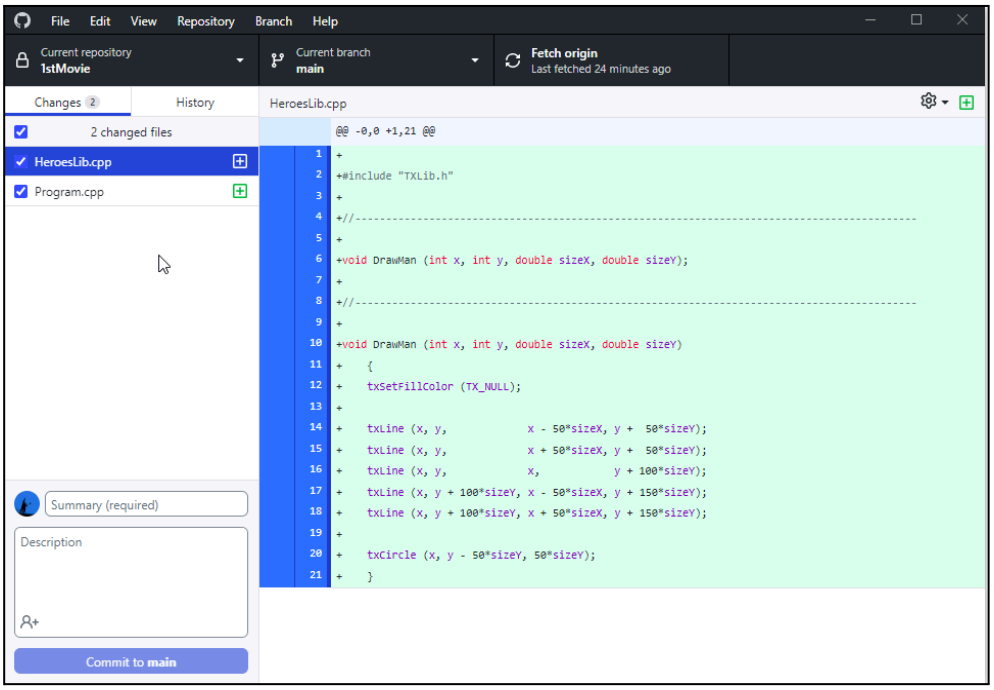
Обратите внимание, что нужно снять галочку с пометки “Keep this code private”, чтобы ваш репозиторий был виден другим людям.



Теперь можно открыть папку с репозиторием и скопировать в нее ваши рабочие файлы проекта. На данный момент это файлы с исходном текстом программ на языке C++ (тип файлов “CPP File”, расширение файла – .CPP). Открыть папку можно, зайдя в Проводник или более профессиональный файловый менеджер (например, Far Manager), или нажав кнопку “Show in Explorer", или сочетание клавиш Ctrl+Shift+F. Также можно посмотреть, как выглядит репозиторий в облачном хранилище на сайте Github.com (но пока наш репозиторий пустой, смотреть там особо нечего).



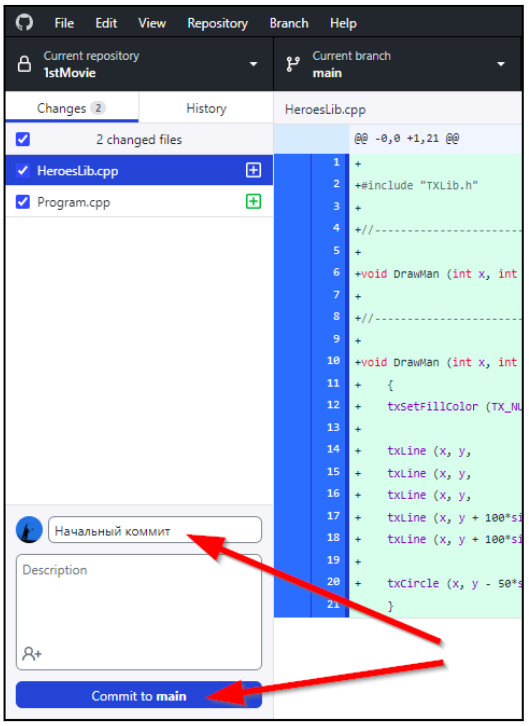
После того, как вы скопируете файлы в папку репозитория, они сразу же станут видны в Github Desktop:

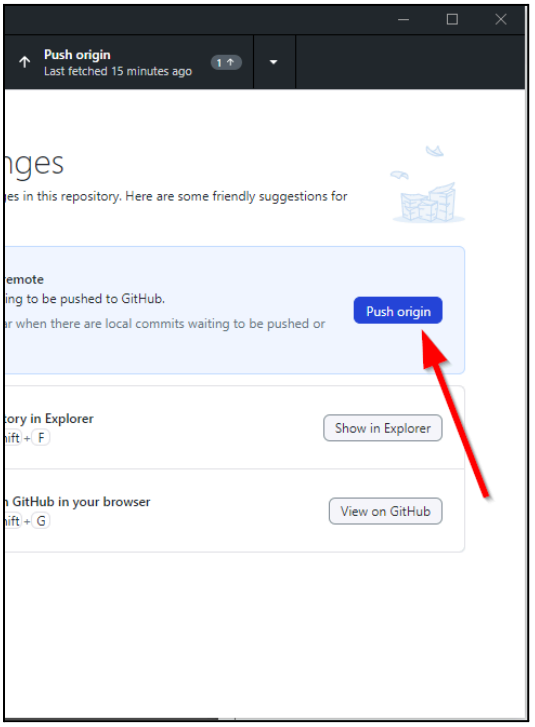


Слева виден список измененных файлов, справа – какие строки в этих файлах изменились. Зеленым цветом и знаком “+” в начале отмечены добавленные строки, красным цветом и знаком “-” в начале – удаленные. Удаленных строк пока нет, потому что мы только что добавили файлы (“все строки всех файлов”) в пустой репозиторий. Если какой-то файл или все файлы какого-то типа не надо сохранять в репозитории, это можно сделать, нажав правую кнопку мыши на нужном файле и выбрав “Ignore file”. Файл при этом перестает показываться в списке файлов в Github Desktop, но в папке на вашем диске он остается. Список игнорируемых файлов лежит в текстовом файле .gitignore (без расширения), его можно редактировать текстовыми редакторами.

Чтобы запомнить все файлы (“сохранить текущее состояние репозитория”), нужно сделать так называемый коммит. Это все равно как нажать Ctrl+S для сохранения файла в редакторе,

только при этом не удаляются прошлые версии файлов. Для коммита надо ввести его название (в Github Desktop это называется Summary), это, как правило, очень краткое описание того, что вы сделали с проектом совсем недавно. Например, вы его только что скопировали в пустой репозиторий. Если нужно более детальное описание, его можно ввести в Description. После этого нажимается кнопка Commit (на рисунке ниже слева), и затем кнопка Push (залить все изменения на сайт Github.com, на рисунке справа):



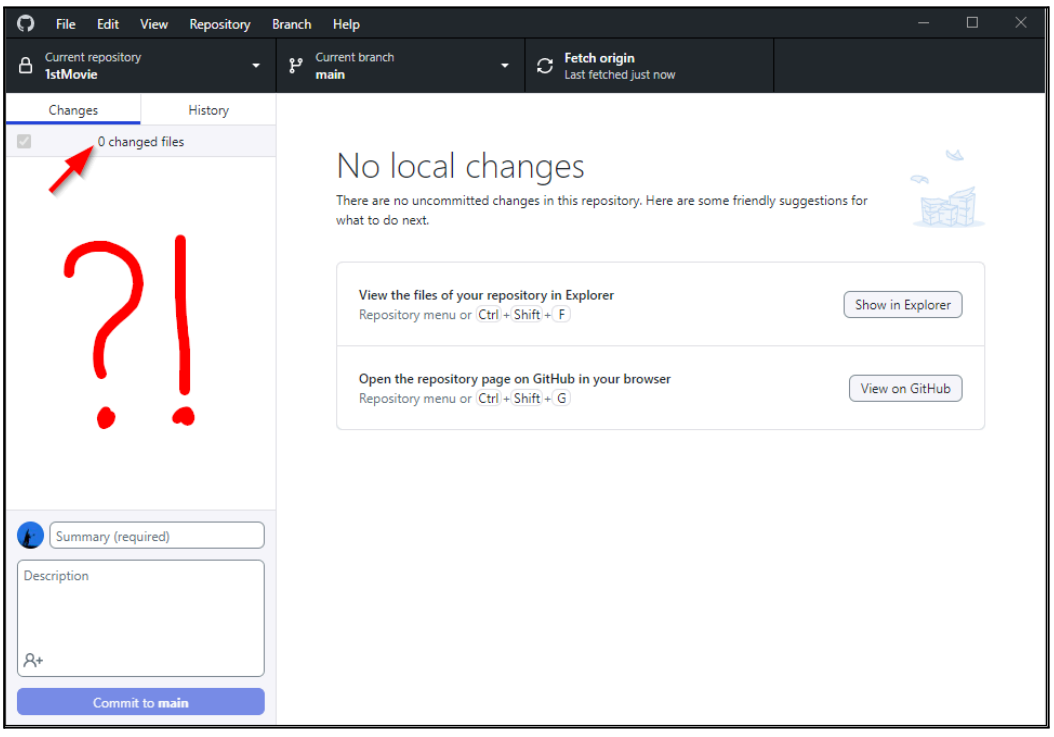


Та-даам... И...

Где файлы?

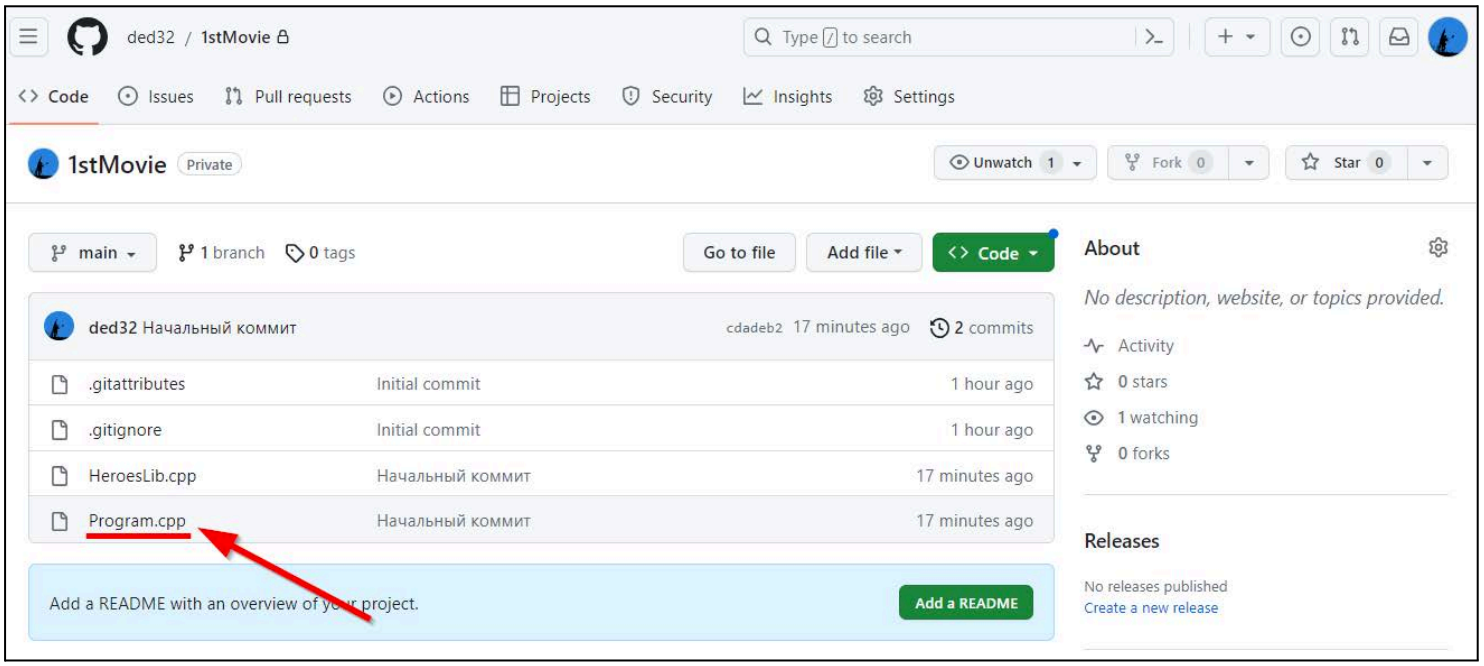
Файлы где???

Где файлы, спрашивается???

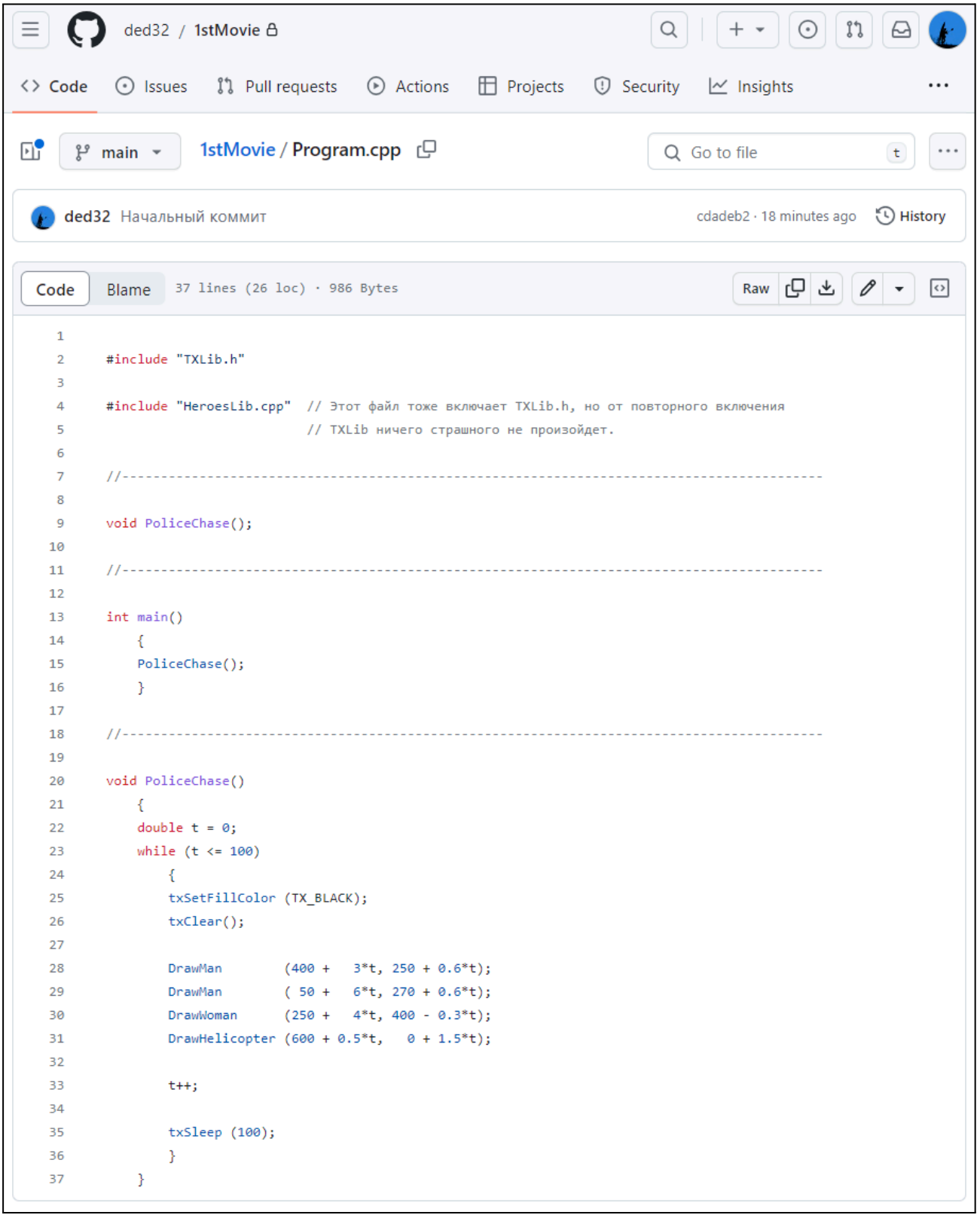


Это что??? ВСЕ ФАЙЛЫ СТЕРЛИСЬ, ДА????? И что, это все, что он делает, этот ваш Гитхаб??? Файлы стирает, да??? :(((((((Больше прогать не буду, пойду погамаю в майнкрафт 100 часов или буду смотреть 200 серий аниме.)

На самом деле все файлы сохранились, причем два раза, в базе данных на вашем диске (после Commit) и в облаке (после Push). Вот как они выглядят в облаке на сайте Github.com (нажимаем кнопку View on GitHub):



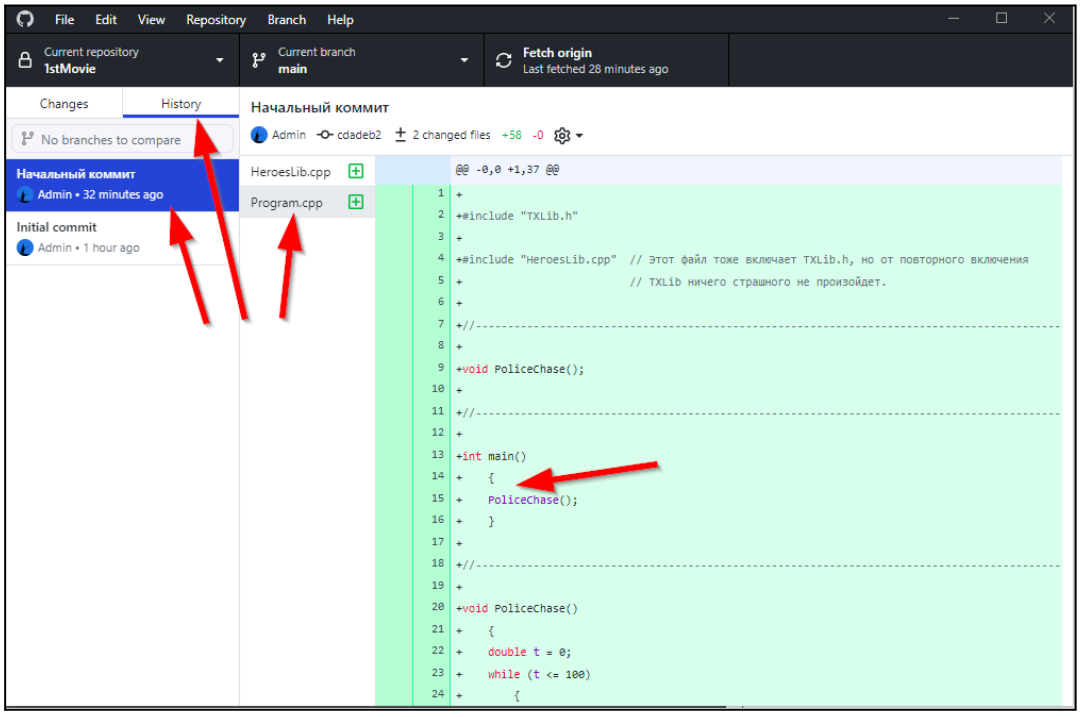
И содержимое файлов сохранилось, уфф (ауф):



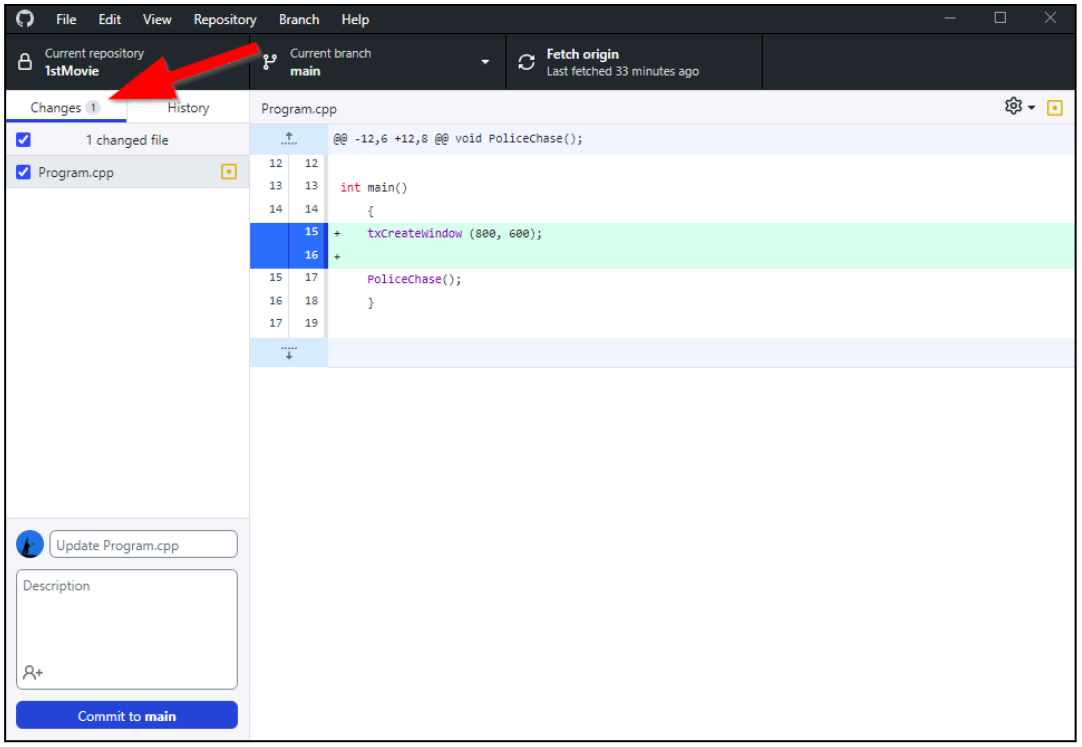

```
1
2   #include "TXLib.h"
3
4   #include "HeroesLib.cpp"  // Этот файл тоже включает TXLib.h, но от повторного включения
5                             // TXLib ничего страшного не произойдет.
6
7   //-----------------------------------------------------------------------------------------
8
9   void PoliceChase();
10
11  //-----------------------------------------------------------------------------------------
12
13  int main()
14      {
15      PoliceChase();
16      }
17
18  //-----------------------------------------------------------------------------------------
19
20  void PoliceChase()
21      {
22      double t = 0;
23      while (t <= 100)
24          {
25          txSetFillColor (TX_BLACK);
26          txClear();
27
28          DrawMan        (400 +   3*t, 250 + 0.6*t);
29          DrawMan        ( 50 +   6*t, 270 + 0.6*t);
30          DrawWoman      (250 +   4*t, 400 - 0.3*t);
31          DrawHelicopter (600 + 0.5*t,   0 + 1.5*t);
32
33          t++;
34
35          txSleep (100);
36          }
37      }
```

Все на месте. На диске при просмотре через Проводник или Far Manager все файлы тоже видны. Почему же в Github Desktop ничего нет?

Потому что Github Desktop отображает не все файлы, а только измененные с момента последнего коммита. Об этом говорит название вкладки Changes в левом верхнем углу программы. Если файлов в проекте много (а это обычное состояние проекта в программировании), и при этом выводить список всех файлов, то будет велик и с ним будет неудобно работать. Если мы переключимся в режим History, то увидим список коммитов слева, а, нажав на коммит, просмотрим входящие в него файлы справа:



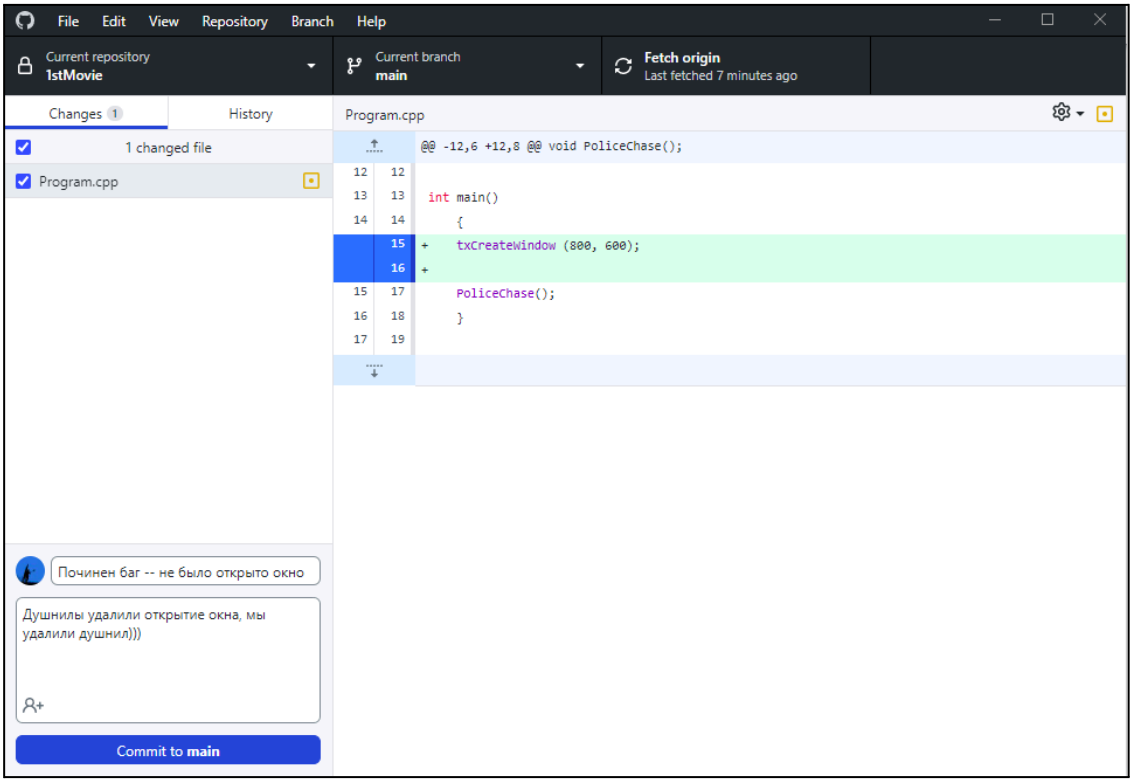
А теперь самое интересное. Обратите внимание, что программа-то не работает: увлекшись репозиториями, мы забыли создать окно рисования с помощью txCreateWindow. Давайте исправим это дело, зайдем в редактор, добавим вызов txCreateWindow, сохраним файл в редакторе, проверим, что с программой все ОК и снова перейдем в Github Desktop, во вкладку Changes:



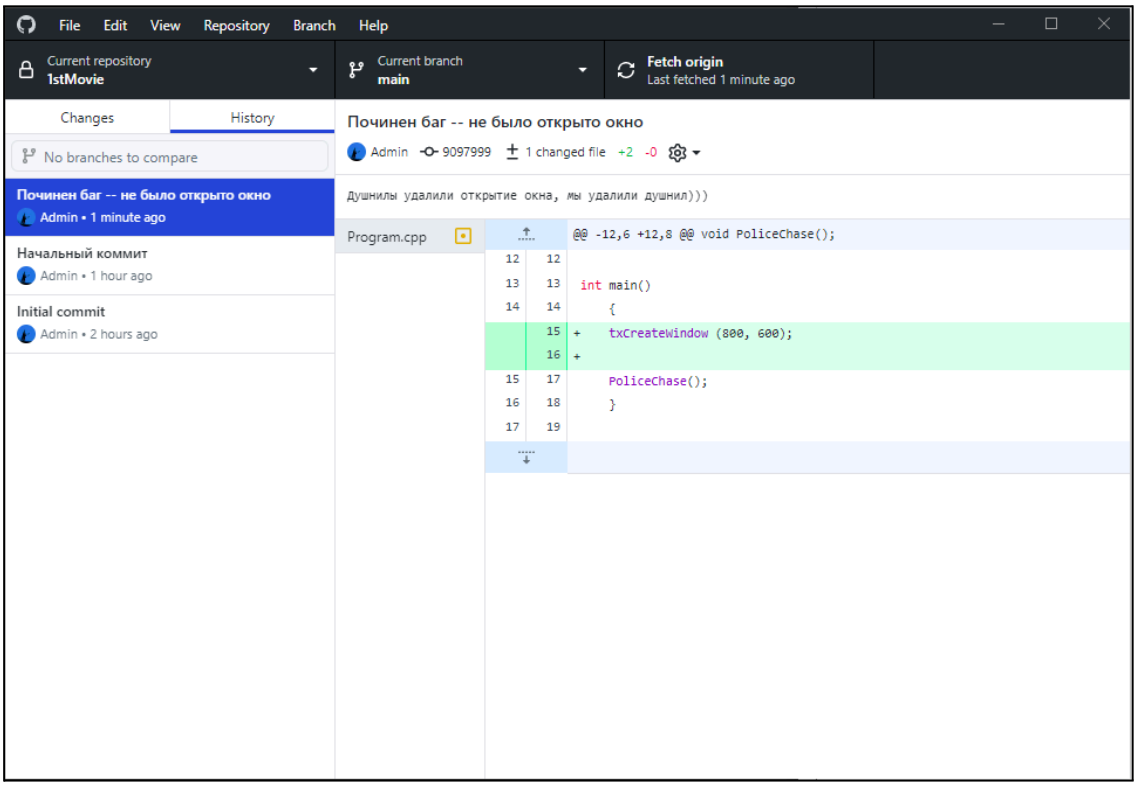
Слева виден список измененных файлов (у нас изменен всего один файл), справа – список изменений в файле: показываются только измененные строчки и ближайшие соседние. Сразу видно, что добавлены вызов txCreateWindow и пустая строка. Все прекрасно.

Если мы теперь сделаем еще один коммит (давайте же сделаем его скорее, мы же исправили программу, а вдруг у нас сейчас отключится свет, сломается комп, сдохнет Интернет, или приключится еще что похуже, а у нас исправление только на нашем компе, а не в облаке!

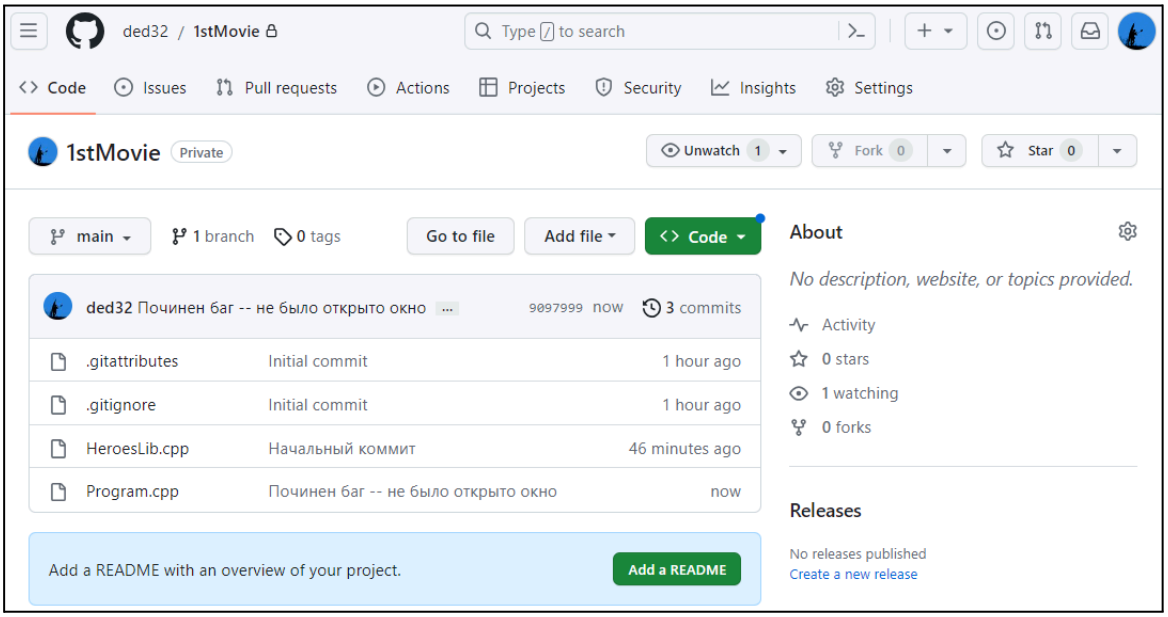
СРОЧНО сохранять в облако, то есть делать коммит, а потом push!). Какое название коммита? Ну мы же починили программу, вставив открытие окна, вот и напишем про это.



Во вкладке History сверху у нас появился новый коммит, справа список файлов в нем и список изменений в файлах:

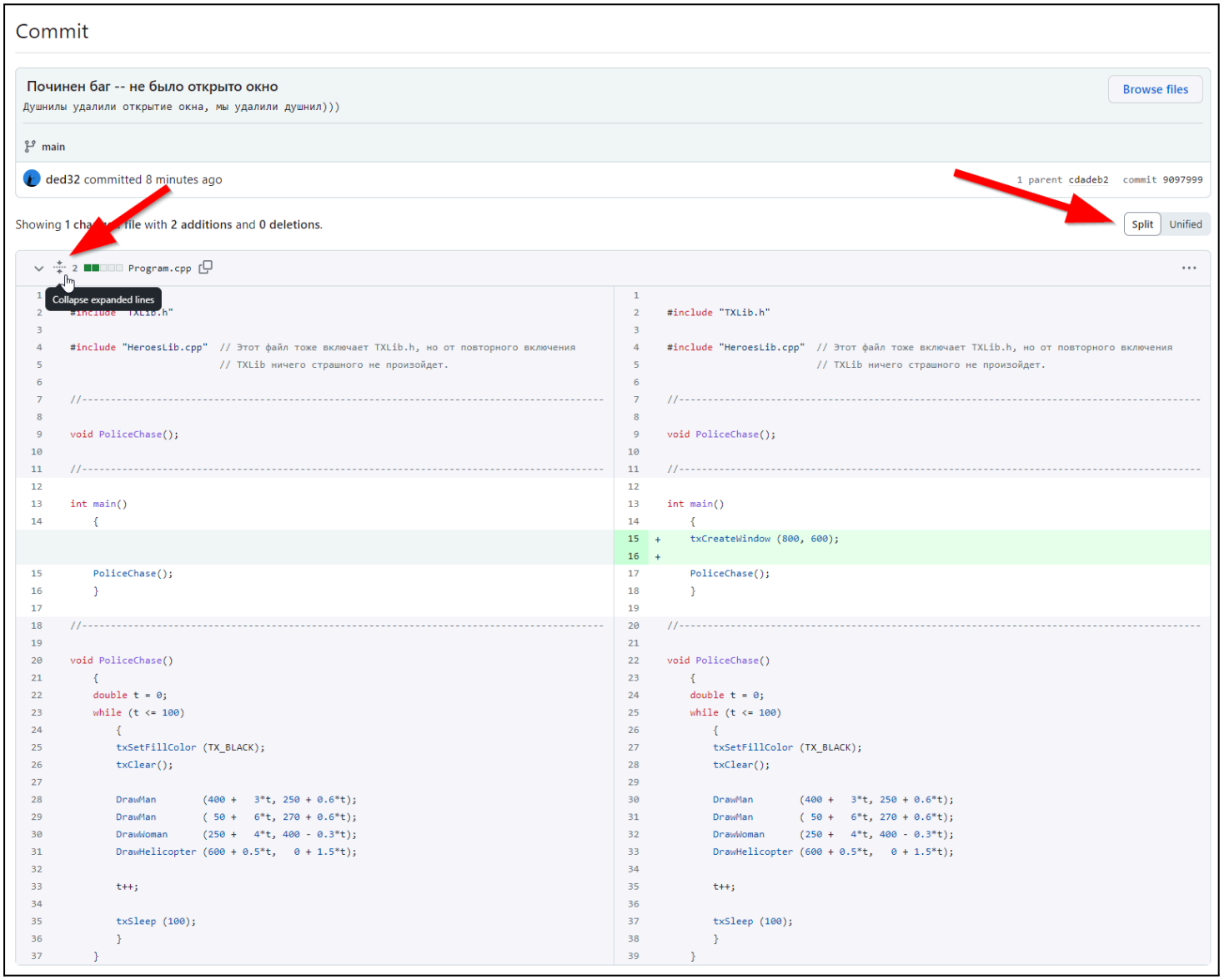


После команды Push на сайте будет новое состояние репозитория:



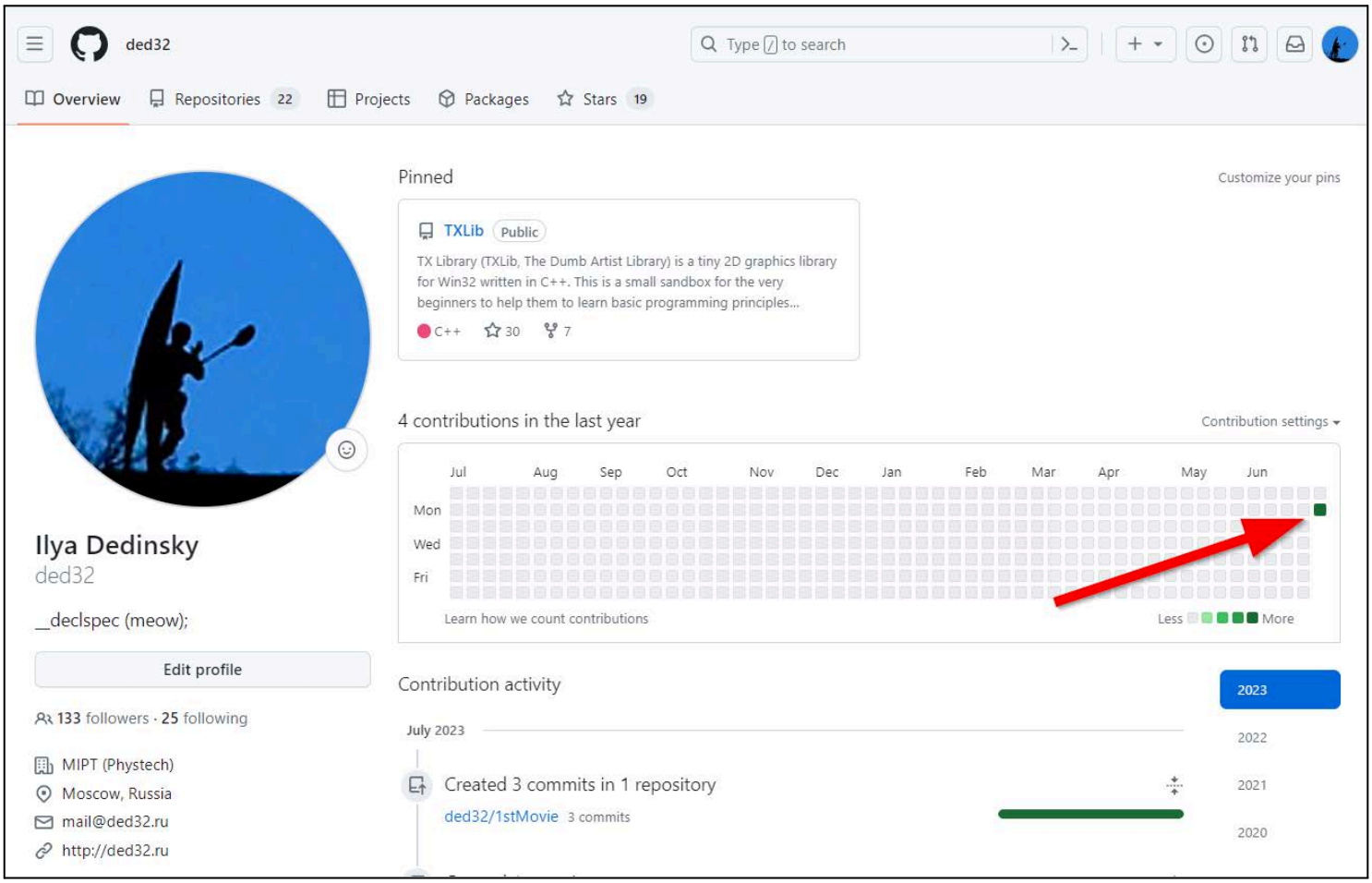
Видно, что файл HeroesLib.cpp так и остался неизменным с момента первого коммита, а файл Program.cpp изменен по причине починки бага. Отлично.

Если нажать на название коммита справа от имени файла, можно посмотреть изменения в развернутом виде (нажмите на кнопки и переключатели, помеченные стрелочками):



Слева видно состояние файла до коммита, справа – после коммита. Этот способ показа называется "diff" (от "difference"). Так и говорят – "Починил? Покажи дифф." Он широко используется в программировании.

Но самое-то главное, самое-то главное-то:



Та-дааам! Мы получили ачивку. На главной странице вашего профиля на Гитхабе отображается ежедневная активность за год, и там видно, что сегодняшний день мы провели не зря – неплохо поработали. Так что, как говорится, жизнь надо прожить так, чтобы не было мучительно стыдно за халтурно написанный код, чтобы не жег позор за лень и белые квадратики на Гитхабе. Видите этот маленький, но гордый зеленый квадратик? Это начало большого пути начинающего профессионального программиста!

...Ну пафос это, конечно, хорошо (временами), но вот что думает про системы контроля версий мировая мемосфера:

Как видите, при пожаре советуют покидать горящее здание в третью очередь, после коммита и пуша[3]. (Все-таки не следует следовать этому совету напрямую, спасайтесь сразу, жизнь людей ценнее кода.)

1.10. Последние штрихи к мультфильму

Прежде всего убедитесь, что у вас все работает и вы сделали коммит в репозиторий с последними изменениями.

Теперь самое время добавить к мультфильму последние штрихи. Во-первых, это движущиеся титры в начале фильма и в конце. Движение вы делать уже умеете, а надписи можно выводить функциями из раздела "Работа с текстом" в Справочной системе TXLib. Здесь не будет их списка и ссылок на раздел – посмотрите сами, это часть задания. Программист обязан быть самостоятельным. :)

После этого, как вы понимаете – коммит.

Титры можно также "нарисовать" из линий. Это более трудоемко и не так красиво, как вывод с помощью txTextOut или txDrawText, особенно если подобрать хороший шрифт с помощью txSelectFont. Однако рисование букв с помощью линий (или других фигур) дает интересные возможности по динамическому изменению букв. Давайте сделаем функцию, рисующую букву "H":

```
void DrawH (int dnc_damage)
    {
    txSetColor (TX_YELLOW, 7);

    txLine (30,                 100,              30 - dnc_damage/2, 170);
    txLine (30 + dnc_damage/4, 135 + dnc_damage, 75 - dnc_damage/4, 135 - dnc_damage);
    txLine (75,                 100,              75 + dnc_damage/2, 170);
    }
```

Что делает параметр "dnc_damage[4]"? Если он равен нулю, то буква "H" сохраняет свою изначальную форму (проверьте). Если он отличается от нуля, то форма буквы искажается. Это легко увидеть, если написать короткую программу, рисующую эту букву с помощью цикла, где damage меняется от 0 до 100, как мы это делали в функциях движения выше. Не забудьте только стирать с экрана черным цветом, иначе рисунки буквы на разных оборотах цикла наслоятся друг на друга. Особенно эффектно выглядит анимация, когда damage не увеличивается, а уменьшается до нуля: в конце движения буква как бы становится "сама собой", "собирается".

Вот пример еще одной функции, искажающей букву "E":

```
void DrawE (int damage, int red, int green, int blue)
    {
    txSetColor (RGB (red, green, blue), 7);

    txLine (100 + damage/2, 100 + damage/2, 100 - damage/3, 170 + damage/3);
    txLine (100 - damage*2, 100 - damage*4, 135 - damage*2, 100 + damage*2);
    txLine (100 + damage,   135 - damage/3, 135 + damage/2, 135 - damage/2);
    txLine (100 - damage/3, 170 + damage*2, 135 + damage*3, 170 - damage*2);
```

[3] Кстати, знатоки git, определите, в чем этот мем неправилен.
[4] "dnc" означает "do not copy", "не копируйте", а лучше называйте параметры своими словами. Осваивайте примеры через их рефакторинг.

```
    }
```

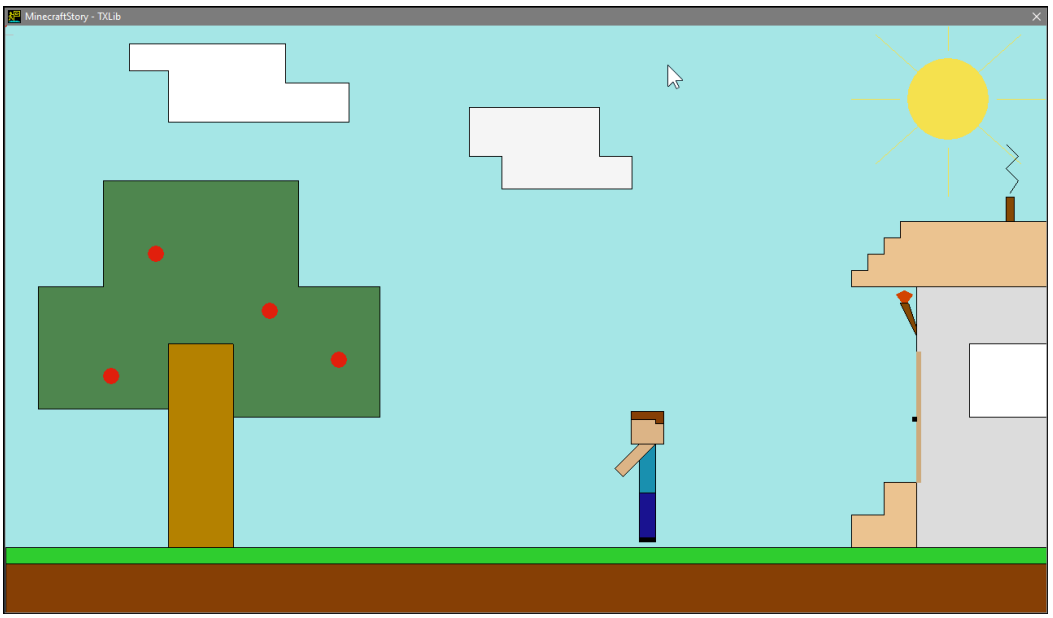
Здесь параметр damage случайным образом прибавляется и отнимается от координат, от чего буква будет меняться непредсказуемо и "разваливаться" в ходе анимации. Параметры red, green и blue задают цвет буквы, что также позволяет менять его в цикле по различным законам.

Буквы также можно двигать независимо друг от друга, если сделать им параметры координат, как мы делали выше, сделать толщину линий параметром и т.д. Все это позволит сделать интересную анимацию, и не только букв, но и персонажей. И закоммитить это на гитхаб.

Также можно добавить к мультфильму музыку или звуки. Функцию для этого посмотрите в Справочной системе TXLib самостоятельно. Внимательно читайте описание и все замечания к нему, TXLib не умеет воспроизводить многие популярные аудиоформаты. Почему? Потому что базовая библиотека Windows, на которой TXLib основан, не поддерживает эти форматы. Однако есть другие библиотеки, которые это умеют, вы можете попробовать их, однако это будет не так просто, как с "тупым художником". Либо можно преобразовать форматы нужных вам аудиофайлов к формату, поддерживаемому TXLib.

И, разумеется, все это выложить (а точнее, регулярно коммитить по ходу работы) – вы знаете куда. С хорошим readme с картинками.

Примеры мультфильмов, сделанных школьниками, можно посмотреть в разделе "Examples" в той папке, куда вы установили библиотеку TXLib, или в репозитории TXLib на гитхабе.



Мультфильм по мотивам *одной известной игры,* автор – ученик 7 класса

1.11. Что главное?

Первый проект закончен, можно прикинуть, чему мы научились на его примере.

Во-первых – "разделяй и властвуй". Этот принцип применяется в программировании настолько часто, насколько возможно. Функция не помещается на экран – дели на части. Файл велик для понимания – дели на части. Библиотека велика и разнородна – дели на несколько библиотек. Это особенно помогает в больших коллективных проектах, где каждый отвечает за свою часть работы, и внутренние детали каждой части не влияют на другие составляющие проекта.

Оборотная сторона этого принципа в том, что надо логично соединить разделенные части. Не все способы удачны, и наверняка вы это уже замечали. Опыт состоит в том, чтобы осознавать и накапливать удачные способы разделения и соединения.

В целом то, как проект разделен на части и как они соединены, задает его архитектуру. Архитектор проекта – одна из высших должностей в профессии разработчика ПО.

Далее – "пиши ясно". Все всегда на свете может (и будет) понято неправильно, а как следствие – неправильно применено или выброшено. При нарушении этого принципа сам автор кода спустя месяц-другой не в состоянии и разобраться в программе. Часто это выдают за неотъемлемую часть профессии, периодически подшучивая. Но на самом деле это результат неграмотной работы.

Принцип этот совершенно бесконечен, так как невозможно предсказать, что в будущем может быть не понято, понято не так, перепутано или забыто. Поэтому ясный код и полная документация строго обязательны для профессионала.

Следующий принцип – "не копируй". Вместо этого дели на части (функции) и эти функции чуть обобщай, чтобы можно было применить в нескольких местах чуть по-разному. Слегка масштабируй в голове способы применения новой функции, чтобы понять, как ее можно полезно обобщить. В то же время, не переусложняй это обобщение, оно будет трудно для понимания, отладки и последующих изменений. Из хорошо обобщенных функций выстраиваются хорошие библиотеки.

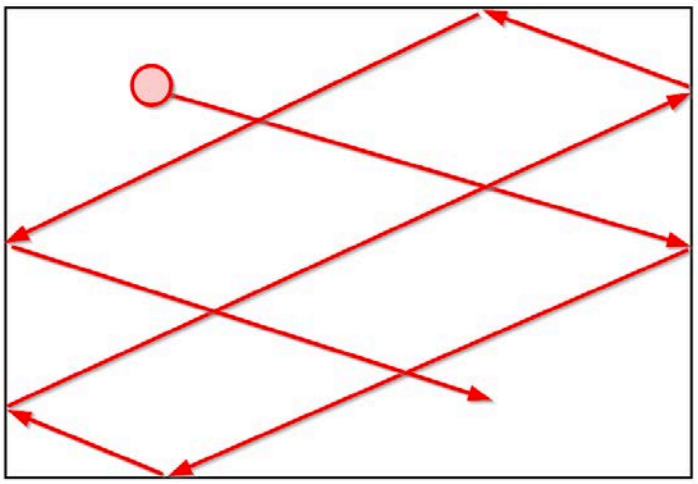
Далее – "делай невидимое видимым". Этот принцип полезен в отладке. Временно выдели мелкие детали ярким цветом, чтобы за ними было удобно следить. Не гадай, чему равны величины в программе, вместо этого сделай ясную красивую распечатку их значений вместе с их названиями. Потрать на это чуть больше времени, но облегчи себе дальнейшую работу над программой. Проверяй, вызвана ли функция вообще, поместив в нее оператор печати или рисование яркого объекта. Человек слаб, и он легко может что-то не заметить. Помогай себе видеть и замечать.

Эти принципы универсальны для разработки любой задачи в программировании, на любых технологиях, языках и аппаратуре. Понимание и следование им гораздо важнее знания конкретного решения, алгоритма, языка и библиотеки. Учиться им приходится всю жизнь до седин, и начать нужно как можно раньше.

2. Второй проект: Игра

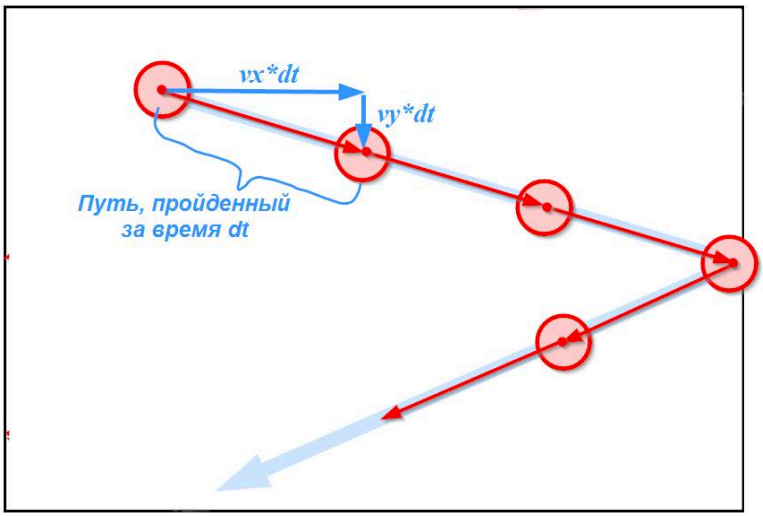
2.1. Дифференциальная модель движения

Задание траекторий движения с помощью формул, зависящих от времени, достаточно мощная вещь, но она быстро становится очень сложной, если траектория движения усложняется. Например, мы хотим построить движение шарика, движущегося по прямой, но отскакивающего от стенок экрана. Вот эта траектория:



Конечно, раз в движении есть повторяемость, и элементом такой повторяемости является движение по прямой, мы можем написать функцию движения по прямой и цикл, который периодически вызывает эту функцию. Для вызова этой функции придется каждый раз рассчитывать координаты точек пересечения траектории движения со стенками экрана. Для прямолинейной траектории это не очень сложно, но если она станет криволинейной, произвольной, это трудно решаемая задача. Есть ли другой способ?

В физике часто встречаются со сложными процессами, в том числе и с движением по сложной траектории, когда рассчитать координаты тела в зависимости от времени “одним расчетом”, отталкиваясь только от начальных координат – сложно. Тогда применяют метод разбиения времени на небольшие интервалы, за каждый из которых тело продвигается на небольшое расстояние. Есть целая наука и даже предмет на Физтехе (называется “численные методы”) о том, как это правильно сделать с наибольшей точностью. Вот версия приведенного выше рисунка, когда траектория разбивается таким образом:



То есть мы:

1) Отказываемся постоянно рассчитывать координаты тела по формулам, зависящим от времени, прошедшего от начала движения (во всех примерах выше мы обозначали его t). Вместо этого разбиваем все движение на фрагменты, каждый длительностью dt (это другая величина, с t не связанная). Создаем переменную для dt. Величину dt можно взять достаточно малой, для начала равной, например, 1;
2) Создаем переменные для координат x и y, которые будем обновлять по ходу цикла движения, а также переменные для скоростей движения vx и vy (то есть проекций вектора движения на координатные оси). Почему переменные? Потому что при столкновении со стенками эти скорости будут меняться на противоположные;
3) На каждом шаге цикла движения будем обновлять координаты тела x и y, считая, что движение внутри фрагмента прямолинейное, меняются при этом только координаты, а скорости постоянны:

```
x += vx * dt;
y += vy * dt;
```

или

```
x = x + vx * dt;
y = y + vy * dt;
```

4) Рисовать тело будем только в начале фрагмента, двигаться оно будет скачками, но при малом dt это будет незаметно;
5) Цикл движения напишем один, и пусть он работает, например, пока не будет нажата какая-то клавиша.

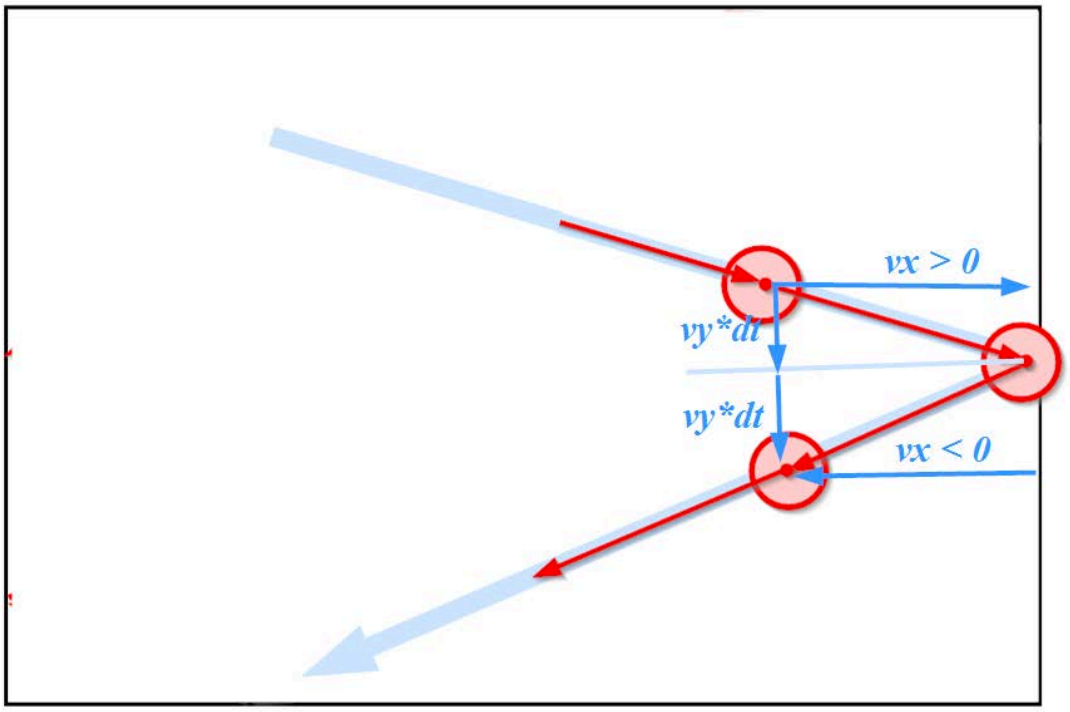
Вот что получилось:

```
void PlayBall()
    {
    double x = 100, y = 100;
    double vx = 7, vy = 5;
    double dt = 1;

    while (! txGetAsyncKeyState (VK_ESCAPE)) // Пока не нажата клавиша Esc. Восклицательный знак -
        {                                    // оператор отрицания. VK_ это "виртуальная клавиша",
                                             // а не "Вконтакте" компании Mail.ru.

        DrawBall (x, y);                     // Предполагаем, что мы заранее написали эту функцию,
                                             // в ней рисуется один шарик без всяких циклов

        x += vx * dt;                        // Обновляем координаты
        y += vy * dt;
```

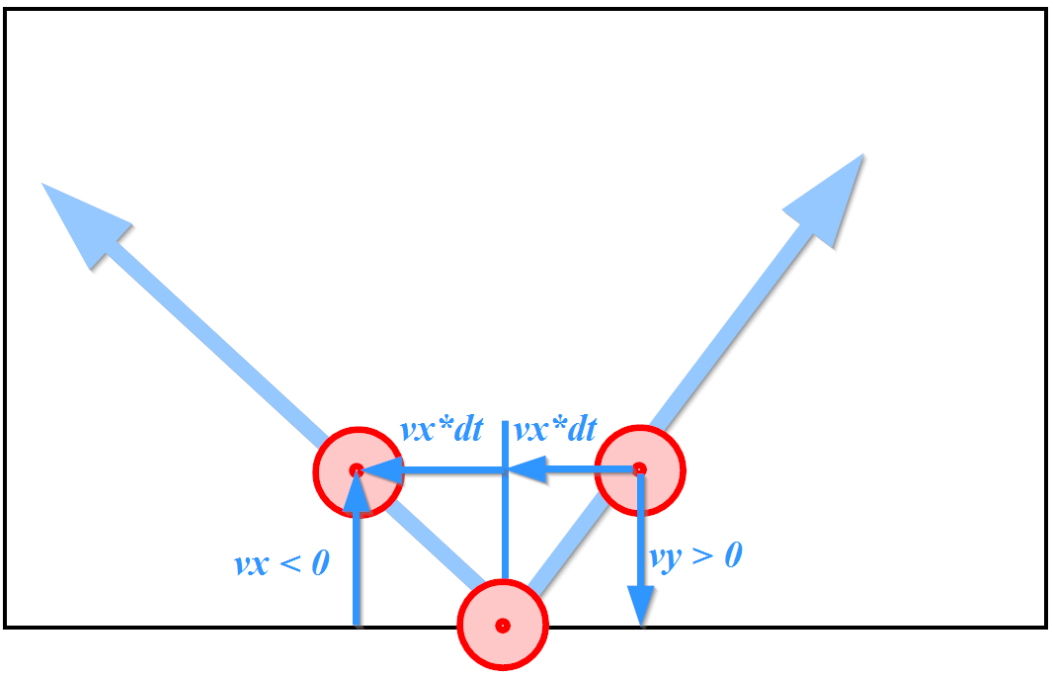
```
        txSleep (10);                    // Небольшая задержка, чтобы не было слишком быстро
        }
    }
```

Пока эффект от этого цикла такой же как обычно: шарик движется по прямой и скрывается за правым краем экрана. Нажимаем Esc - функция завершается.

Теперь добавим изменение скоростей при столкновении со стенками. При отскоке от вертикальных стенок меняется скорость vx:



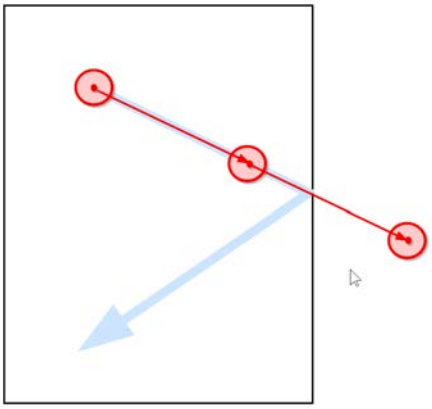
А при отскоке от горизонтальных меняется vy:



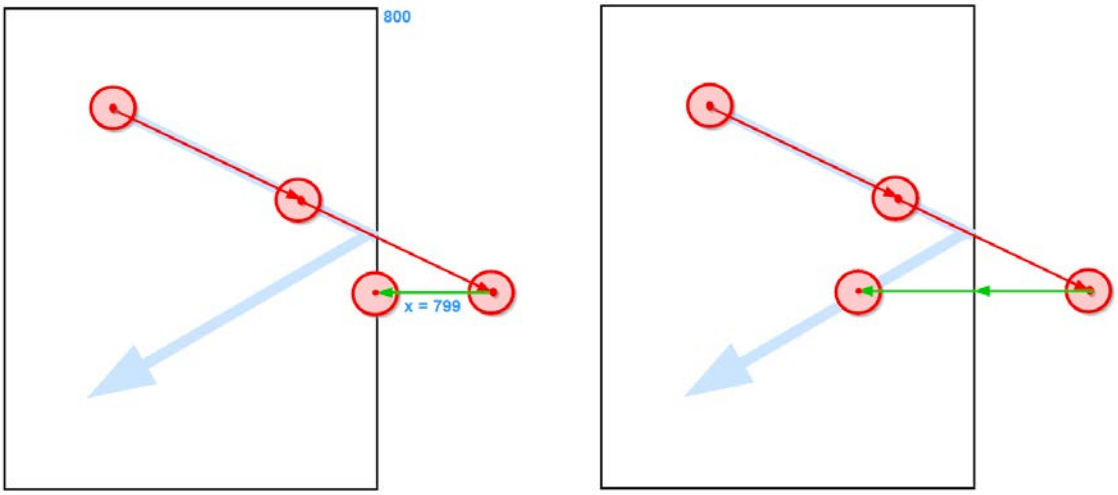
Если у нас абсолютно упругий удар, то абсолютное значение скорости (длина вектора скорости) не меняется, но меняется направление, задаваемое одной из проекций, так, что угол падения равен углу отражения.

Если шарик оказался в углу и произошло взаимодействие с двумя стенками, обрабатываем столкновения независимо.

Так как мы разбили траекторию на прямолинейные фрагменты конечной длины, будет одно затруднение. Предположим, шарик движется слева направо. На каком-то шаге шарик может подойти близко к правой стенке, но пока не столкнуться с ней. На следующем шаге перемещение шарика vx*dt может оказаться достаточно большим, чтобы к концу интервала шарик "въехал" в стенку, то есть новая координата x окажется не равной координате правой стенке, а сразу больше ее. Вот эта ситуация:



Есть три пути решения этой ситуации. Один – корректировать ту координату шарика, которая "вылезла" за пределы экрана, чтобы она обязательно была внутри, то есть "насильственным" присваиванием задавать ей значение, чуть меньшее координаты стенки (на рис. ниже слева). Второй – разделять интервал dt на две части, до столкновения и после, далее находить координаты точки пересечения со стенкой и рассчитывать вторую часть интервала dt, отталкиваясь от этой точки. Третий следует из геометрии ситуации столкновения (на рис. ниже справа):



Для простоты мы пойдем первым путем, то есть будем просто корректировать одну из координат. (Ценители математики могут попробовать остальные варианты самостоятельно.) Пусть правая стенка стоит на координате 800, а нижняя на 600. На псевдокоде это получается так:

```
...
(тут мы изменили значение координат)
...
```

```
    если (координата x правее правой стенки, то есть больше 800), то
        изменить значение скорости: vx = -vx;
        откорректировать координату: x = 799.

    если (координата y ниже нижней стенки, то есть больше 600), то
        изменить значение скорости: vy = -vy;
        откорректировать координату: y = 599.
```

Аналогично для левой и верхней стенок:

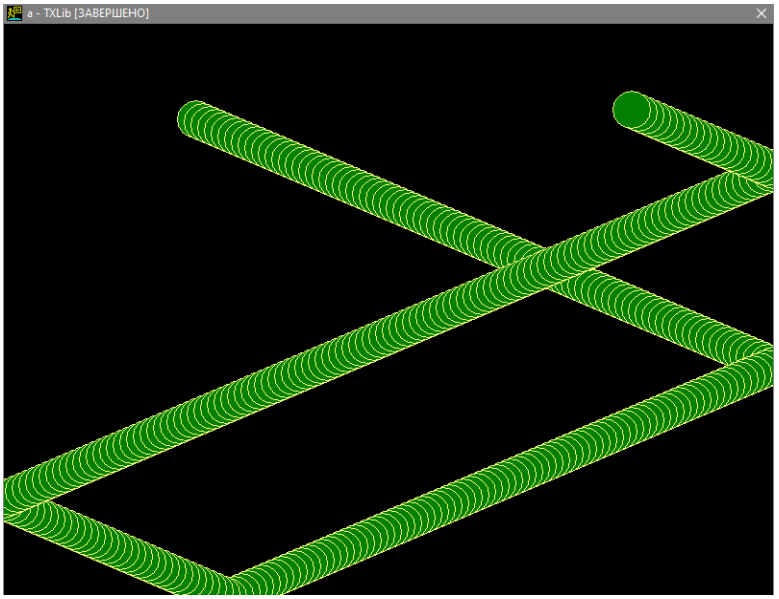
```
    если (координата x левее левой стенки, то есть меньше 0), то
        изменить значение скорости: vx = -vx;
        откорректировать координату: x = 1.

    если (координата y выше верхней стенки, то есть меньше 0), то
        изменить значение скорости: vy = -vy;
        откорректировать координату: y = 1.
```

Некоторым, возможно, покажется, что после vx = -vx (и vy = -vy) шарик по-прежнему полетит налево (вверх), так как выражения -vx (-vy) дают отрицательные скорости. Но если вспомнить, что при срабатывании этих условий скорости vx и vy уже были отрицательны, так как шарик уже летел влево (вверх), то "минус на минус даст плюс" с результирующие скорости будут больше нуля.

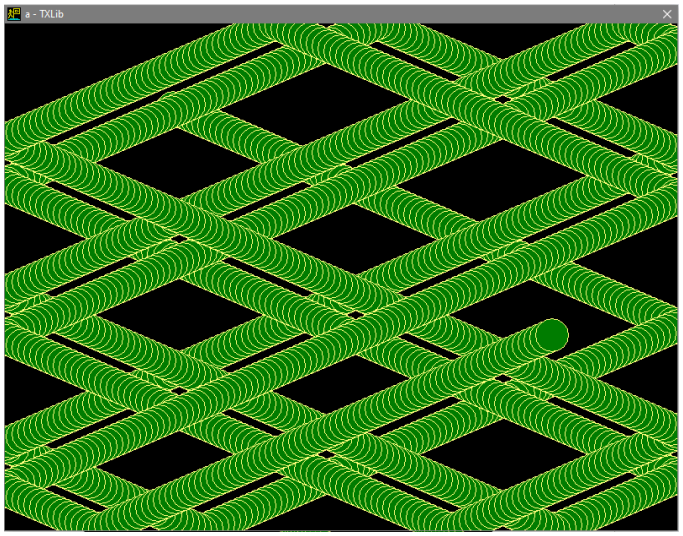
Что получится в итоге (не надо это копипастить, напишите сами!):

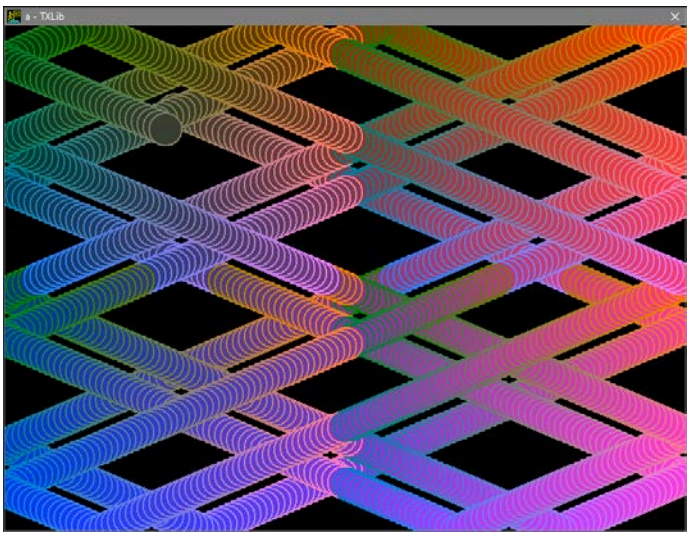
```
int main()
    {
    txCreateWindow (800, 600);

    int x = 200, y = 100, vx = 7, vy = 3;
    int dt = 1;

    while (! txGetAsyncKeyState (VK_ESCAPE))
        {
        DrawBall (x, y);

        x += vx * dt;
        y += vy * dt;

        if (x >= 800) { vx *= -1; x = 800; }
        if (x <=   0) { vx *= -1; x =   0; }
        if (y >= 600) { vy *= -1; y = 600; }
        if (y <=   0) { vy *= -1; y =   0; }

        txSleep (10);
        }
    }
```

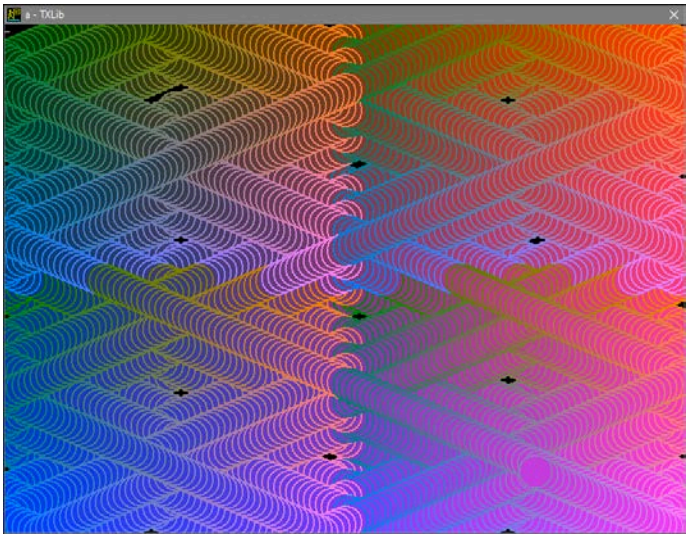
Поскольку шарик не стирается, то видна вся его траектория и получается вот что:



Через некоторое время экран заполняется таким вот трубопроводом, который можно раскрасить, указывая компоненты цвета шарика в зависимости от координат и/или скоростей:







Попробуйте, вам понравится. :)

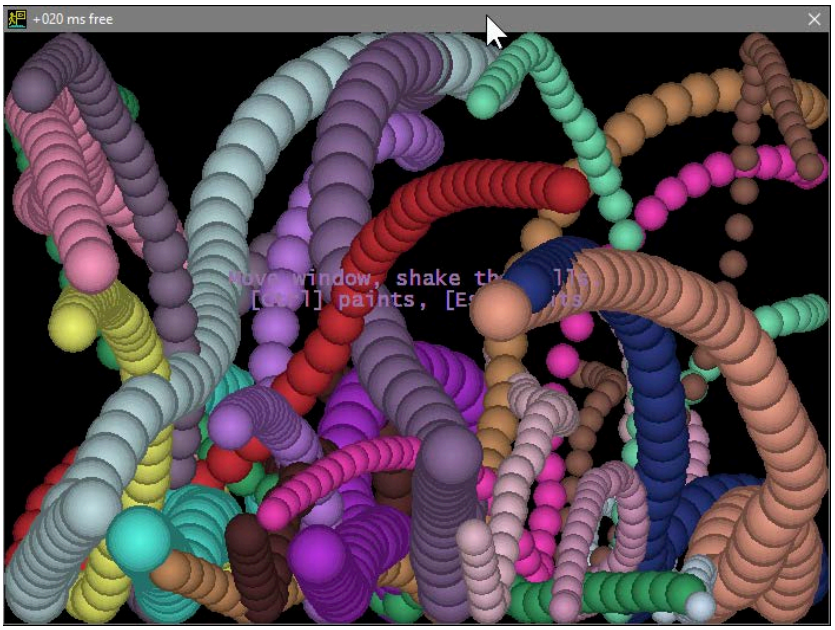
Что еще можно добавить в модель? Вспомним, что в физике бывает еще и равноускоренное движение. Добавим переменные, содержащие проекции вектора ускорения (ax = 0 и ay = 1) и изменение скорости под воздействием силы, приводящей к ускорению движения:

```
vx += ax * dt;
vy += ay * dt;
```

или

```
vx = vx + ax * dt;
vy = vy + ay * dt;
```

Запустите. Интересно? А это мы еще про второй закон Ньютона и массу не вспомнили. Срочно зовите физиков, будем рассчитывать полет на Марс, пока Илон Маск нас не опередил!



Траектории движения нескольких шаров под действием силы тяжести. Из примеров к библиотеке TXLib, автор – ученик 7 класса.

2.1.1. Задание

А вы думали, будет без него? Без заданий неинтересно. Задание будет на масштабирование: сделайте одновременное движение трех (и только трех) шариков разных цветов и одинакового размера. Для выполнения совершенно достаточно материала этой главы, не надо привлекать дополнительных источников.

После выполнения вам кое-что покажется не вполне удобным, это нормально. Три шарика большого неудобства не вызовут. Вперед, пишите код!

2.2. Указатели

(Вы написали программу в задании?)

.

.

.

.

.

.

.

.

.

.

.

.

(Точно написали?)

.

.

.

.

.

.

.

.

.

.

.

(А если проверю?)

.

.

.

.

.

.

.

.

.

.

...

Ну что ж, значит, действительно написали. Смотрите, шарики бегают, картинка глаз радует, но код неудобен. Давайте разбираться, мы уже знаем, что с неудобным кодом не надо мириться, надо искать архитектурные решения – и начинаем, конечно, с “разделяй и властвуй”.

Что здесь муторного? Физика. И в основном, отскоки от стенок. Четыре стены, три шарика, двенадцать (двенадцать, Карл!) почти одинаковых условных операторов. Неприятно. Но физика у трех шариков одинаковая, и поэтому давайте выделим физику одного шарика в функцию, и будем передавать в нее координаты, скорости и ускорения этого шарика. Вызовем ее трижды, по одному разу для каждого шарика. Вуаля, у нас уже снова четыре условия вместо двенадцати. Делаем:

```
void MoveBall (int x, int y, int vx, int vy, int dt)
    {
    x += vx * dt;
    y += vy * dt;

    if (x >= 800) { vx *= -1; x = 800; }
    if (x <=   0) { vx *= -1; x =   0; }
    if (y >= 600) { vy *= -1; y = 600; }
    if (y <=   0) { vy *= -1; y =   0; }
    }
```

Вызываем:

```
int main()
    {
    ...

    int x = 200, y = 100, vx = 7, vy = 3;
    int dt = 1;

    while (...)
        {
        ...
        DrawBall (x, y);

        MoveBall (x, y, vx, vy, dt);
        ...
        }
    }
```

Код прекрасен. Запускаем. Та-дааааам…

…

…

…

Вот гадость. Шарик стоит на месте.

Почему? Мы же все сделали правильно.

Оказывается, это довольно фундаментальный вопрос. И связан он с тем, что раньше мы передавали в функции параметры, которые не менялись внутри функции. То есть функция принимала, скажем, переменную x, но эта переменная менялась не внутри функции, а снаружи

ее, в цикле, который находился в более внешней функции. Например, так были устроены все функции рисования объектов мультфильма, и так же устроена и функция DrawBall. (Точнее, эти параметры вполне могли меняться внутри функций, но все дело в том, что после выхода из функций эти изменения были нам не важны и мы их могли игнорировать. Теперь это не так: при выходе из функции MoveBall координаты и скорости просто обязаны измениться, ведь в этом весь смысл функции физики.)

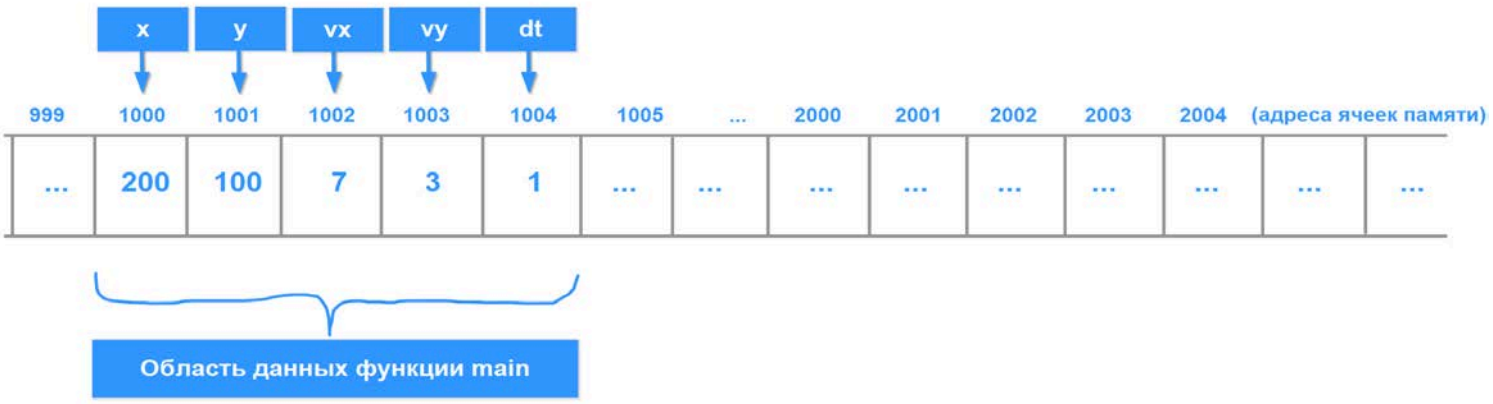
Что же в действительности происходит при передаче параметров?

В действительности вот что: при передаче в функцию значения параметров **копируются**.

Это кажется тривиальным, но у этого принципа много неожиданных следствий:

1) При передаче параметра в функцию для него заводится отдельная переменная, с отдельным месторасположением в оперативной памяти компьютера. Это происходит даже если имена передаваемой переменной и параметра совпадают – имена здесь не играют никакой роли. То есть, если вы передаете в функцию, например, значение переменной x, и соответствующий параметр функции тоже называется x, то это будут два разных икса, две разные переменные.
2) Передаваемое значение копируется в эту новую переменную. То есть в начале работы функции параметр x будет равен значению передаваемого x.
3) Если параметр функции не меняется внутри функции, то тот факт, что мы работаем не с “оригиналом”, а с копией, остается незамеченным. Но если копия меняется, то к оригиналу это изменение отношения не имеет.
4) Все параметры функции при выходе из функции уничтожаются, их значения пропадают. Поэтому, если копия изменилась, мы возвращаемся к “оригиналу”, не изменившему свое значение.

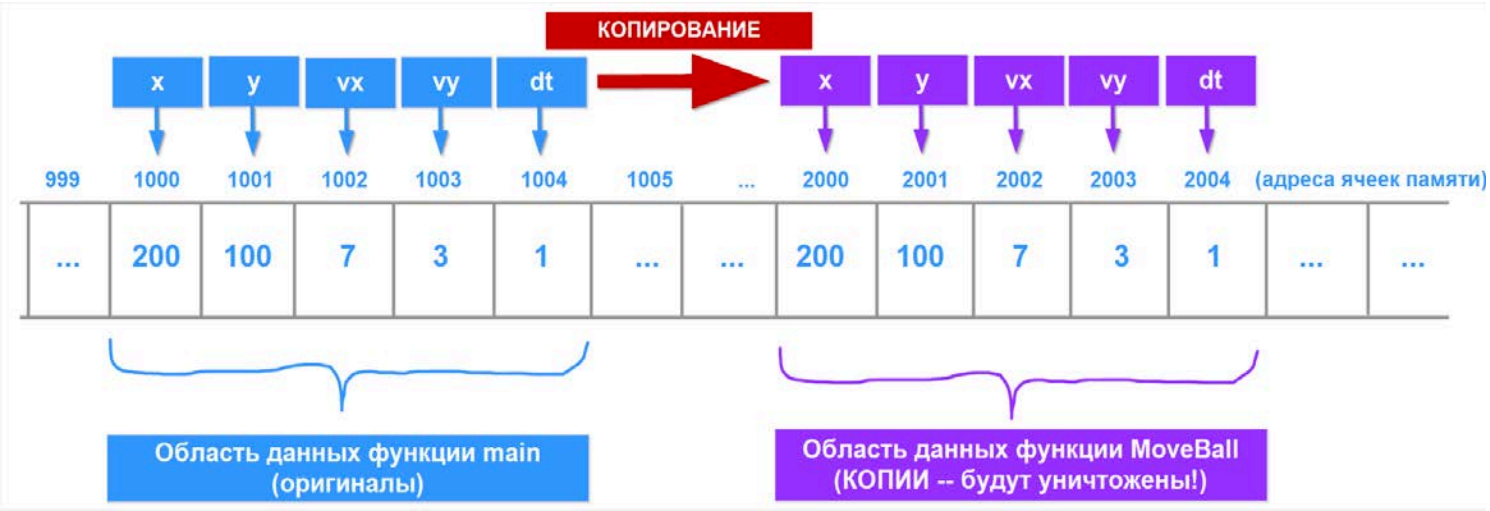
Вот поясняющие рисунки. Вот как приблизительно выглядит оперативная память компьютера до вызова функции MoveBall:



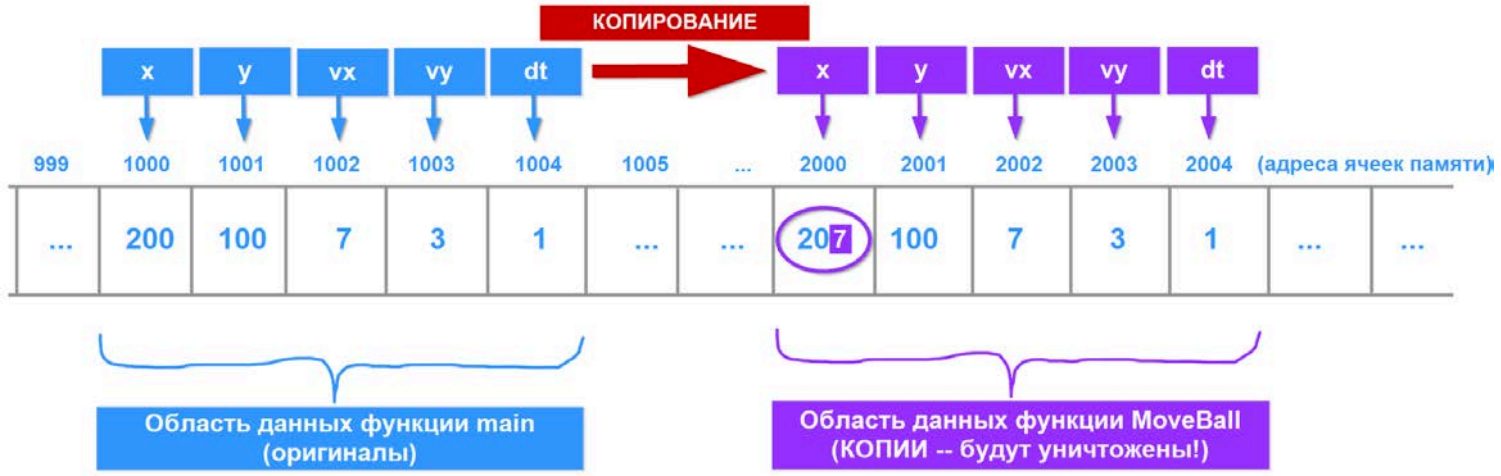
Оперативная память состоит из байтов – ячеек, хранящих числа. У каждой ячейки есть ее порядковый номер, или адрес. На рисунке адреса ячеек памяти условны, в действительности каждая целочисленная переменная обычно занимает четыре ячейки памяти (четыре байта), поэтому в действительности адреса будут кратны четырем. Однако для нашего примера это несущественно.

Все переменные каждой функции располагаются в определенной области памяти, а названия переменных – это просто синонимы их адресов. Так сделано потому, что человеку проще работать с названиями, чем напрямую с адресами.

На следующем рисунке видим, что происходит при вызове функции MoveBall. Передаваемые переменные (параметры функции) копируются в другую область памяти, и поэтому справа мы видим вторые экземпляры каждой переменной, которые появились в процессе этого копирования:



Функция MoveBall может работать только с переменными, принадлежащими ей, в правой части рисунка. Но они существуют только во время работы этой функции, и после выхода из нее они уничтожатся. Вот что происходит при попытке изменить наш икс внутри функции:



Изменяется не тот икс. Не main-овский, а MoveBall-овский. А этот икс, как и все переменные MoveBall справа, будет уничтожен при выходе. При возврате в main мы увидим, что оригинальный икс не изменился. Вот гадость.

Вот как такая ситуация выглядит в коде:

```
void PlayBall()                         // PlayBall - вызывающая функция
    {
    int x = 100, ...;                   // Здесь создается переменная x, принадлежащая функции
                                        // PlayBall. Будем обозначать ее как PlayBall::x. Она
                                        // изначально равна 100.

    int vx = 7, ...;
    int dt = 1;
```

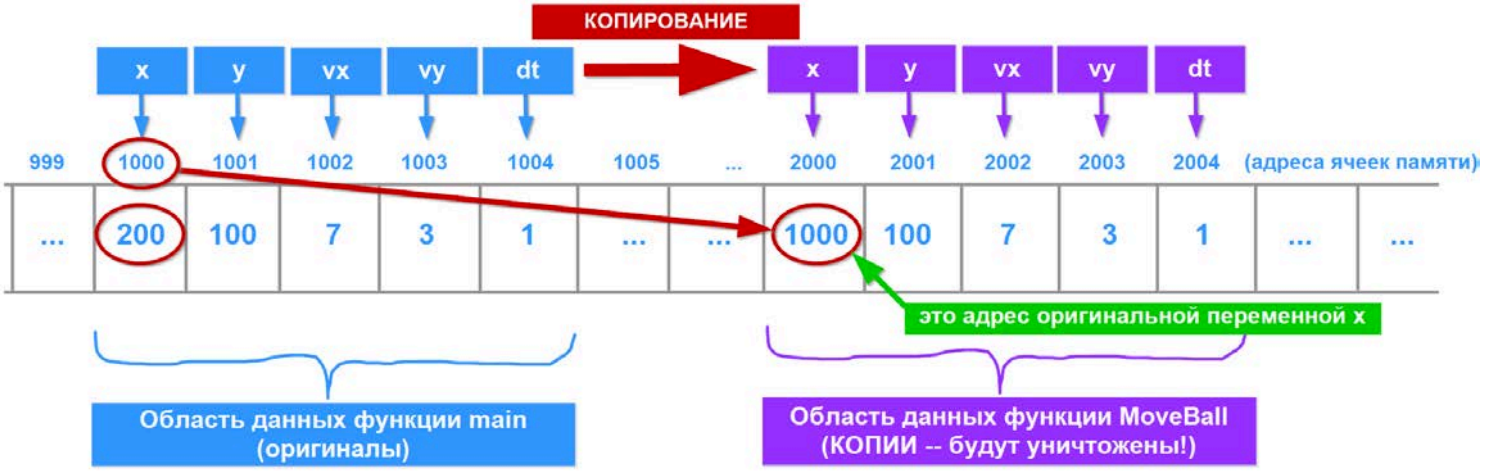
```
    while (...)
        {
        DrawBall (x, ...);              // Нарисуем шарик два раза: до и после вызова MoveBall,
                                        // для контроля. Шарик появится в координате x = 100.

        ...                             // Здесь PlayBall::x передается в функцию MoveBall.
        MoveBall (x, ..., vx, ..., dt); // Пусть идет первый оборот цикла. Тогда она равна 100.
                                        // Теперь идите в функцию MoveBall и смотрите, что будет.

        ...                             // Здесь мы вышли из MoveBall, копия (MoveBall::x) пропала,
                                        // и мы снова работаем с оригиналом (PlayBall::x), который
                                        // по-прежнему равен 100

        DrawBall (x, ...);              // Шарик рисуется вновь в координате x = 100, и движения
        ...                             // мы не видим
        }
    }


                                        // Это MoveBall - вызываемая функция

void MoveBall (int x, ..., int vx, ..., int dt) // При ее вызове создалась новая переменная-копия x,
                                                 // принадлежащая функции MoveBall (т.е. MoveBall::x
                                                 // и в нее скопировалось число 100
    {
    ...
    x += vx * dt;                       // Здесь MoveBall::x изменилась, стала равной 107.
    ...                                 // Но это копия, а оригинал PlayBall::x остался
    ...                                 // равным 100
    ...
    ...                                 // Здесь функция MoveBall завершилась и переменная
    }                                   // MoveBall::x (копия) с новым значением 107
                                        // уничтожилась. Остался только оригинал PlayBall::x
                                        // со старым значением 100. :((
```

Если это не очень понятно, вы можете поиграть в такую игру. Пусть один из вас, например, Петя, сыграет роль вызывающей функции PlayBall, а, например, Миша – роль вызываемой функции MoveBall. Петя возьмет листок бумаги и крупно напишет на нем x = 100. Это будет переменная x, принадлежащая его функции PlayBall, равная 100. Дальше Петя вызовет Мишу, то есть функцию MoveBall. При вызове Миша тоже возьмет листок бумаги и крупно напишет на нем свой x, заинтересованно посмотрит на Петин листок с иксом и скопирует его значение себе: напишет у себя x = 100. В этот момент в классе будут два листка с иксами: один Петин (PlayBall::x, оригинал), другой Мишин (MoveBall::x, копия), и пока они неотличимы. Дальше Миша изменит свой x с 100 на 107 (не стирая, перечеркнет 100 и напишет 107), и в этот момент значения копии и оригинала разойдутся. Вроде бы x Миши более новый и более полезный, но так как это всего лишь копия, при возврате из MoveBall она уничтожится – Миша демонстративно порвет свой листочек с полезным значением 107 ("так не доставайся же ты никому"). Мы вернемся к Пете, который будет грустно держать в руках листок со старым значением "x = 100" без всякого 107. Шарик, подчиняющийся Петиной переменной PlayBall::x, останется стоять на координате 100.

Ок, мы поняли, что проблема в копировании. Можно ли отменить копирование? Можно, и для этого вызывающая функция PlayBall должна передавать в вызываемую функцию MoveBall не копию значения оригинальной переменной, а информацию, где в памяти компьютера лежит эта самая оригинальная переменная PlayBall::x. Эта информация – номер ячейки памяти компьютера, где хранится значение переменной, или, как говорят, адрес переменной. Конкретное значение этого адреса нам знать не нужно, главное, чтобы он был передан в

вызываемую функцию MoveBall вместо копии значения[5]. Тогда вызываемая функция может "залезать" по этому адресу и менять там нужное значение.

Вот такая схема работы на картинке. При передаче параметров мы передадим в функцию MoveBall не копию значения икса (200), а адрес оригинальной переменной, где он хранится (ячейка номер 1000):



Теперь у функции MoveBall хранится адрес оригинального икса, то есть номер ячейки памяти, где он лежит. Пользуясь этим адресом, мы "залезем" по адресу, получим оригинальный икс и изменим его. Для доступа к оригинальной переменной используется специальный оператор * (звездочка), который приписывают к адресу. Можно считать, что это оператор "сходить в гости к другу", если у вас есть адрес этого друга. Вот схема работы в этом случае (шаги 1 - 2 - 3 - 4):

На рисунке оригинальный икс показан уже измененным. На картинке, может быть, это выглядит немного сложно, потому что там много пояснений. Следуйте по шагам 1-2-3-4. В коде будет проще.

Чтобы перевести этот способ в игру, надо представить, что листочки со значениями не валяются просто так на столах, а являются пронумерованными листами толстой книги. На каждом листе этой книги написано какое-то одно значение, и его можно получить или изменить, во-первых, зная нужный номер страницы и, во-вторых, открыв книгу на этой странице. Тогда переменные это просто синонимы номеров страниц, и пользоваться ими

[5] Хотя, если интересно, его можно распечатать на экране вызовом printf ("Переменная x лежит в ячейке номер %p\n", &x); или printf ("Адрес переменной x равен %p\n", &x);

удобнее лишь потому, что нам удобнее работать с именами, а не с адресами. (Чтобы не портить книгу изменениями, можно вложить в нужное место листок с переменной. Ну или найти в книге опечатку.)

В этой схеме Петя (вызывающая функция PlayBall) передает Мише (вызываемой функции MoveBall) не копию значения переменной, а ее адрес, то есть номер страницы, где написано значение переменной. Пусть этот номер страницы 42, тогда Петя передает Мише не число 100, а число 42, трактуемое как адрес. Каждый раз, когда Миша (вызываемая функции MoveBall) захочет узнать или изменить значение, написанное на странице, он должен открывать книгу на странице 42, то есть осуществлять доступ к памяти компьютера. Тогда он сможет изменить значение оригинальной Петиной переменной PlayBall::x, даже в том случае, если Петя отвернулся, заснул и не видит Мишиных действий. Такое "залезание" на страницу нужно делать каждый раз, когда требуется доступ к оригинальной Петиной переменной. (Если вам кажется, что это занимает чуть больше времени, то это правда.) В нашем примере Петя "залезает" на страницу 42, чтобы увеличить координату со 100 до 107. По возвращении из Мишиной функции MoveBall, Петя (функция PlayBall) увидит на странице под номером 42, где расположена его переменная x, долгожданное обновленное значение 107. Извините за душнильство, но пришлось, чтобы вы не запутались.

Как узнать значение адреса? Для этого в языке Си есть специальный оператор & (читается "амперсанд")[6]. Его надо приписать слева от имени переменной. Теперь вызов MoveBall будет выглядеть так:

```
    MoveBall (&x, ..., vx, ..., dt);
```

И это все, что нужно сделать в вызывающей функции для переменной x.

Так. Это полдела. Теперь с этим адресом надо правильно обойтись в вызываемой функции, MoveBall. Раньше мы в ней принимали координату x в форме значения, теперь будем принимать в форме адреса.

Тут два момента: во-первых, надо изменить все операции с переменной x, добавив к ним оператор "залезания" по адресу (доступа к ячейке памяти). Работать непосредственно с номерами ячеек в данном случае бессмысленно: например, прибавляя к адресу перемещение vx * dt, мы получим адрес другой ячейки памяти, в которой лежит какая-то другая информация, никак не связанная с переменной x. Доступ к ней и даже изменение ее технически возможны, но последствия непредсказуемы: от полного игнора до зависания программы или закрытия ее по критической ошибке. Это называется неопределенным поведением, undefined behaviour или UB, и является одной из самых больших неприятностей.

Оператор "залезания в ячейку" (доступа к памяти) обозначается звездочкой *, и это не та звездочка, что используется для умножения. Нужная нам звездочка ставится перед переменной, заданной в адресной форме: *x. То есть выражения изменения координат меняются с

```
x += vx * dt;
```

или

```
x = x + vx * dt;
```

на

[6] Можно объявить конкурс на лучшее написание символа &. Не у всех получается с первого раза. И это вы еще буквы "кси" не видели. Привет от кафедры высшей математики Физтеха. :)

`*x += vx * dt;` или `*x = *x + vx * dt;`

Обратите внимание, что звездочки пишутся каждый раз, когда нам нужно получить доступ к значению переменной, то есть перейти от адресной формы к значению.

Во-вторых, надо изменить описание параметра x в прототипе и заголовке функции MoveBall, чтобы объяснить компилятору, что теперь параметр x является адресом. Делается это так:

```
void MoveBall (int* x, ..., vx, ..., dt) ...
```

Выражение “int* x” (иногда пишется как “int *x”) читается справа налево как “икс является адресом целого числа”, а в более традиционном направлении слева направо – “целого числа адресом икс является”, что намекает на связь языка Си с магистром Йодой и объясняет его феноменальную мощь в программировании.

Функция MoveBall теперь будет выглядеть как

```
void MoveBall (int* x, int y, int vx, int vy, int dt)
    {
    *x += vx * dt;  // *x вместо просто “x”, т.к. x - теперь адрес, а не значение координаты
     y += vy * dt;  // С “y” пока оставим все как было

    ...
    }
```

Сделайте все эти изменения (возможно, у вас переменные объявлены как double, так что не копируйте текст книги напрямую), и у вас шарик должен поехать вправо...

...по горизонтали. Но не по вертикали. Потому что то же самое нужно сделать и с переменной y.

...но не отражаясь от стенок. Потому что при отражении скорости меняются (помните условия отражени?). Поэтому нужно... ну вы поняли.

Но нужно ли все переменные переводить в адресную форму? А вот и нет. Во-первых, доступ по адресам замедляет программу. Во-вторых, если какая-то функция, для каких-то своих служебных внутренних целей, захочет изменить значение параметра, он изменится не только у нее, но и у вызывающей функции. А это нарушит логику ее работы, и ошибку такого рода придется искать очень долго. Ну вот буквально: вызываешь функцию MoveBall, написанную коллегой, а она в некоторых случаях резко уменьшает dt, например, для большей точности отталкивания шарика. Если dt передается в форме адреса, то уменьшится “главный” dt функции PlayBall, управляющий всем движением, и это вряд ли то, что мы хотели. Ошибку, конечно, найдем, но через денек-другой. Эх, ночи отладки, красные глаза! Короче, не надо гнать все через адреса. Программа надежнее будет.

Напоследок о терминологии. Здесь мы использовали слово “адрес” и “адресная форма”, потому что это понятнее и ближе к тому, что на самом деле происходит в компьютере. Однако в программировании более часто используются слова “указатель” (“pointer”) и производные от него. В табличке ниже приведены синонимы для терминов:

Указатель	Адрес, “номер ячейки”
Передача по указателю	Передача параметра в форме указателя или адреса

Передача по значению	Передача параметра в форме копии значения
Взятие указателя	Получение адреса переменной с помощью оператора & (амперсанд)
Доступ по указателю, разыменование указателя	Доступ по адресу, “Залезание в ячейку с определенным адресом”

В некоторых языках есть еще так называемые ссылки, но в больших проектах они часто запутывают код. Указатели используются более явно и поэтому гораздо яснее.

2.2.1. Задание

Оно простое – на рефакторинг. Возьмите вашу программу с тремя шариками и гигантским циклом с двенадцатью ифами внутри и примените там функцию движения с указателями. Функция PlayBall должна резко сократиться, код стать кратким и ясным, а вы – получить удовольствие. :)

2.3. Управление шариком

Движением шарика можно управлять. Выше мы пользовались функцией txGetAsyncKeyState, которая определяет, нажата ли та или иная клавиша. На ее основе можно сделать контроль шарика с клавиатуры. Это несложно: каждую клавишу (стрелка вверх, вниз, влево, вправо или привычные многим ‘W’, ‘A’, ‘S’, ‘D’) можно проверить отдельным условным оператором, их будет не менее четырех, а потом возникнет желание управлять еще и другим шариком, операторов станет больше... надеюсь, вы поняли, что лучше сразу делать функцию, тогда будет действительно несложно. Так как функция будет менять значение скоростей шарика, то передадим их ей по указателю (то есть, в форме адреса). Получится:

```
void ControlBall (int* vx, int* vy);

void PlayBall()
    {
    int x = 100, y = 100;
    int vx = 7, vy = 5;
    int dt = 1;

    while (! txGetAsyncKeyState (VK_ESCAPE))
        {
        DrawBall (x, y);

        MoveBall (&x, &y, &vx, &vy, dt);

        ControlBall (&vx, &vy);

        txSleep (100);
        }
    }

void ControlBall (int* vx, int* vy)
    {
    if (txGetAsyncKeyState (VK_LEFT))  *vx--;
    if (txGetAsyncKeyState (VK_RIGHT)) *vx++;
```

```
    if (txGetAsyncKeyState (VK_UP))    *vy--;
    if (txGetAsyncKeyState (VK_DOWN))  *vy++;
    }
```

Вот гадость! Опять не работает, шарик движется, но не управляется.

Причина – в приоритете операторов, то есть в том, в каком порядке компилятор рассматривает операторы. В выражениях вида `*vx++` два оператора: оператор разыменования (доступа по адресу) `*`, и оператор увеличения, стоящий после имени переменной, ++. Оператор ++ имеет больший приоритет и рассматривается первым, а оператор * – вторым, хотя должно быть наоборот: сначала мы должны были получить доступ к переменной, а потом ее увеличить. Однако из-за неудачных в данном случае приоритетов мы увеличиваем адрес, что в нашем случае бессмысленно. Спасти положение, как и во всей математике мира и вселенной в целом, помогут скобки:

```
    if (txGetAsyncKeyState (VK_RIGHT)) (*vx)++;
```

И все дела.

Если вы захотите изменить клавиши управления, то в Интернете несложно найти константы кодов клавиш, которые принимает GetAsyncKeyState, по запросу “VK_ESCAPE” или названию любой другой такой константы.

Что будет, если мы захотим управлять одновременно двумя шариками? Посмотрите на код:

```
void PlayBall()
    {
    ...

    while (...)
        {
        ...
        ControlBall (&vx1, &vy1);
        ControlBall (&vx2, &vy2);
        ...
        }
    }
```

Клавиши управления для обоих шариков будут одинаковыми, и независимого управления не получится.

То есть нам надо повысить гибкость функции ControlBall, чтобы она реагировала и на другие клавиши, а не только прописанные в ней VK_LEFT, VK_RIGHT и другие, говоря научно – промасштабировать ее по клавишам управления. Если для каждой клавиши управления сделать параметр, и передавать для одного шарика один набор кодов клавиш, а для другого – другой, то получится управлять независимо:

```
void PlayBall()
    {
    ...

    while (...)
        {
        ...
```

```
        ControlBall (&vx1, &vy1, VK_LEFT, VK_RIGHT, VK_UP, VK_DOWN);
        ControlBall (&vx2, &vy2, 'A',     'D',      'W',   'S');
        ...
        }
    }
```

Функция ControlBall при этом будет иметь прототип

```
void ControlBall (int* vx, int* vy, int keyForLeft, int keyForRight, int keyForUp, int keyForDown);
```

названия параметров тут, возможно, слегка детские (из-за "For") и длинные, измените их, чтобы было чуть короче, но обязательно, чтобы осталось так же ясно. Посоветуйтесь с коллегами, какие имена подойдут.

2.4. Возврат значения из функции

Передача параметров в функцию позволяет, в основном, передать данные в одном направлении: из вызывающей функции в вызываемую. Конечно, есть передача по указателю, когда переданное значение можно менять. Но это не всегда удобно, особенно если такое значение только одно, и ради него не хочется делать указательный параметр.

Вот бы как в математике: там каждая функция возвращает какое-то значение. Так устроены, например, синус, косинус, логарифм – мы пишем y = sin (x), подразумевая, что мы передаем в функцию x, а она высчитывает внутри себя значение синуса и передает его "наружу", и оно попадает в переменную y. Такой способ записи краток и удобен, почему бы его не использовать в программировании?

Такое ожидание в языке каких-то возможностей по аналогии с другими языками или науками вполне логично. Языки программирования создают люди, близкие к какой-то вычислительной науке, а это математика, физика или их комбинации. Поэтому обозначения из этих наук часто переходят, иногда чуть изменяясь, в языки программирования.

Поэтому вполне логичен ваш вопрос к человеку, знающему какой-то язык, как в нем записать то-то и то-то, по аналогии с тем-то и тем-то в другом языке или науке. Если у вас есть такой знакомый, то изучение программирования пойдет быстрее. Только надо точно формулировать вопрос, а не то знакомый ответит не на то, что вы спрашивали, даст непонятный ответ или перепишет за вас кусок программы, в котором вы потом ничего не поймете.

Итак, сейчас мы хотим сделать вот что.

Предположим, у нас есть фрагмент программы, в котором мы рассчитываем расстояние между шариками по теореме Пифагора, и если оно меньше суммы радиусов (здесь это просто число 20), то обнаруживаем столкновение и ненадолго перекрашиваем фон окна в красный цвет:

```
...
if (sqrt ((x1-x2)*(x1-x2) + (y1-y2)*(y1-y2)) < 20)
    {
    txSetFillColor (TX_RED);
    txClear();

    txSleep (100);

    txSetFillColor (TX_BLACK);
```

```
        txClear();
        }
    ...
```

Выражение это не самое короткое, и при этом потенциально подобные расчеты могут быть сложнее. В таких случаях, конечно, их выносят в отдельную функцию. Но как вернуть из функции результат расчета? Конечно, можно объявить переменную и передать ее указательным параметром:

```
    double distance = 0;
    CollideBalls (x1, y1, x2, y2, &distance);
    if (distance < 20)
        {
        ...
        }
```

Но это не очень удобно. Вместо этого можно объявить функцию CollideBalls так:

```
double CollideBalls (double x1, double y1, double x2, double y2)
    {
    double distance = sqrt ((x1-x2)*(x1-x2) + (y1-y2)*(y1-y2));
    return distance;
    }
```

Это означает, что в результате своих расчетов функция вычисляет результат, который может быть возвращен вызывающей стороне. Тип возвращаемого значения – double, это видно в начале заголовка. Оператор return заканчивает выполнение функции ("выходит" из нее, пропуская лежащие ниже операторы) и указывает, какое именно значение надо вернуть. После возврата этот результат (возвращаемое значение) может быть присвоен какой-либо переменной или сразу подставлен в выражение или условие:

```
    double distance = CollideBalls (x1, y1, x2, y2, &distance);
    if (distance < 20)
        {
        ...
        }
```

или просто

```
    if (CollideBalls (x1, y1, x2, y2, &distance) < 20)
        {
        ...
        }
```

Сравните с "если sin (x) < 0.5, то ...". Здесь такой же смысл.

Когда срабатывает оператор возврата значения return, например, return 5, то его аргумент (число 5) помещается во временную область хранения данных – ячейку памяти или регистр процессора, из которой вызывающая сторона впоследствии может взять значение (5) и использовать для разных целей. Например, присвоить какой-то переменной (distance), сравнить с каким-то числом (20), стать частью арифметического выражения и т.д. При срабатывании оператора return функция немедленно прерывает свою работу и возвращается

обратно к вызвавшей ее функции, и никакие другие операторы той функции, где сработал return, уже не исполнятся.

Что означает тип возвращаемого значения void ("пустой"), который мы писали почти везде? Это значит, что в такой функции нет никакого математически осмысленного значения, которого можно было бы вернуть. Поэтому мы там ничего (void) и не возвращали. В самом деле, наши функции в основном рисовали объекты и математические результаты там не образовывались, возвращать было нечего: не возвращать же количество закрашенных пикселей. В такой функции может стоять оператор return, но без какого либо числа или выражения за ним, и он используется, чтобы срочно выйти из функции, пропустив ее оставшуюся часть.

Единственная функция, которая была не void, была функция main. Она возвращала значение типа int, которое передавалось в операционную систему после завершения нашей программы. Формально это значение игнорируется операционной системой, но часто оно значит что-то очень важное (например, если в алгоритмическом контесте вы вернете ненулевое значение, то проверяющая система не будет проверять выведенный вами ответ и сразу сочтет решение неверным). Вообще в профессиональном мире принято всегда возвращать из main осмысленные значения, потому что от результата работы вашей программы может зависеть работа других программ.

Когда стоит использовать возвращаемое значение, а когда – указательные параметры? Если у функции есть "главный" или единственный результат, например, расстояние между шариками, его лучше сделать возвращаемым значением. Код будет естественней и легче, ближе к привычной математической записи. Если результатов несколько, как, например, в функции расчета физики, то лучше сделать указательные параметры, хотя пользоваться ими менее удобно и более многословно: надо обязательно заводить переменную, передавать по указателю и только потом использовать.

Что будет, если в функции, которая обязана что-то возвратить (то есть она имеет тип возвращаемого значения, отличный от void), забыт оператор return? Это будет очень печально, потому что функция все равно будет считаться возвращающей значение, но оно будет случайным и бессмысленным. Формально это не считается ошибкой, и программу можно будет скомпилировать и запустить, но ее поведение будет случайным (неопределенным), и эти баги придется искать уже в ходе отладки. Как раз тот случай, когда либерализм мешает и хочется, чтобы компилятор нас проверил построже. (Слабо сказать такое на контрольной по русскому? А ЕГЭ по нему входит в сумму баллов поступления... эх!) Для этого существует специальная директива

```
    #pragma GCC diagnostic warning "-Wreturn-type"
```

Если разместить ее в начале программы (даже до директив #include), то компилятор начнет проверять такие вещи и выдавать в качестве предупреждений. Попробуйте специально сделать такую ошибку и посмотрите, что скажет компилятор. Подобных директив много, есть групповые (-Wall и другие), они все очень полезные, и чем новее версия компилятора, тем он больше ловит всяких ошибок. В библиотеке TXLib все эти директивы включены, поэтому вы получаете больше диагностических сообщений, чем без нее, что страхует вас от многих ошибок. Посмотрите на исходный текст TXLib.h и поищите там директивы `#pragma GCC diagnostic`, их там много, а в последующих версиях будет еще больше. :)

2.4.1. Задание

В качестве задания доделайте программу, используя возвращаемые значения, и поуправляйте шариками.

2.5. Игра

Если собрать все вместе, то получается вот что. Сравните с тем, что вы написали, посмотрите различия и обсудите с коллегами. Не всегда примеры в этой книге выглядят так, как автор написал бы на работе, а не в учебном курсе: здесь приходится многое упрощать, чтобы не делать примеры слишком большими. Тем не менее главное – "разделяй и властвуй", разделяй на функции и соединяй их с помощью параметров и возвращаемых значений – тут есть.

```
//-----------------------------------------------------------------------------------------
//! @mainpage
//! Пример простейшей игры, в которую пока нельзя выиграть или проиграть.
//!
//! @par Правила игры
//!      Двое одновременно гоняют каждый свой шарик, управляя разными клавишами.
//!
//! @par Управление
//!    - Первый игрок (желтый шарик)  -- стрелки курсора
//!    - Второй игрок (зеленый шарик) -- клавиши W (вверх), A (влево), S (вниз), D (вправо)
//!
//! @par Список функций
//!      Game1.cpp
//!
//-----------------------------------------------------------------------------------------

#include "TXLib.h"

//-----------------------------------------------------------------------------------------

void PlayBall();
void DrawBall (int x, int y, COLORREF color, COLORREF fillColor);
void MoveBall (int* x, int* y, int* vx, int* vy, int dt);
void ControlBall (int* vx, int* vy, int keyLeft, int keyRight, int keyUp, int keyDown);
bool CollideBalls (double x1, double y1, double x2, double y2);

//-----------------------------------------------------------------------------------------

int main()
    {
    txCreateWindow (800, 600);

    PlayBall();
    }

//-----------------------------------------------------------------------------------------
//! Основная функция игры
//!
//! @todo
//! -# Заменить целочисленные координаты на действительные числа (int -> double)
//-----------------------------------------------------------------------------------------

void PlayBall()
    {
    int x1 = 100, y1 = 100, vx1 =  7, vy1 = 5;
    int x2 = 100, y2 = 100, vx2 = -7, vy2 = 5;
    int dt = 1;
```

```
    HDC black = txCreateCompatibleDC (800, 600);

    txBegin();

    while (! txGetAsyncKeyState (VK_ESCAPE))
        {
        txAlphaBlend (0, 0, black, 0, 0, 0.25); // Это вместо стирания черным прямоугольником.
                                                // Здесь функция txAplhaBlend копирует изображение
                                                // черного фона с учетом прозрачности 25%.
                                                // Разберитесь, как работают функции
                                                // txCreateCompatibleDC, txDeleteDC и txAlphaBlend,
                                                // там много интересного.
        DrawBall (x1, y1, TX_ORANGE, TX_YELLOW);
        DrawBall (x2, y2, TX_GREEN,  TX_LIGHTGREEN);

        MoveBall (&x1, &y1, &vx1, &vy1, dt);
        MoveBall (&x2, &y2, &vx2, &vy2, dt);

        ControlBall (&vx1, &vy1, VK_LEFT, VK_RIGHT, VK_UP, VK_DOWN);
        ControlBall (&vx2, &vy2, 'A',     'D',      'W',   'S');

        if (CollideBalls (x1, y1, x2, y2) == 1)
            {
            txSetFillColor (TX_RED);
            txClear();
            }

        txSleep();
        }

    txEnd();
    txDeleteDC (black);
    }

//---------------------------------------------------------------------------------------
//! Перемещение шарика по законам физики
//!
//! @param x   x-координата шарика
//! @param y   y-координата шарика
//! @param vx  Скорость шарика по оси x
//! @param vy  Скорость шарика по оси y
//! @param dt  Интервал времени движения
//!
//! @note
//!       - Отталкивание приближенное
//!       - Движение в прямоугольнике (0, 0, 800, 600)
//!
//! @todo
//! -# Сделать более точную коррекцию координат при отталкивании от стенок
//! -# Заменить магические числа 0, 0, 800, 600 (координаты) константами
//! -# Заменить целочисленные координаты на действительные числа (int -> double)
//---------------------------------------------------------------------------------------

void MoveBall (int* x, int* y, int* vx, int* vy, int dt)
    {
    *x += *vx * dt;
    *y += *vy * dt;

    if (*x >= 800) { *vx *= -1; *x = 800; }
    if (*x <=   0) { *vx *= -1; *x =   0; }
    if (*y >= 600) { *vy *= -1; *y = 600; }
    if (*y <=   0) { *vy *= -1; *y =   0; }
    }

//---------------------------------------------------------------------------------------
```

```
//! Управление шариком с клавиатуры
//!
//! @param vx         Скорость шарика по оси x
//! @param vy         Скорость шарика по оси y
//! @param keyLeft    Код клавиши для смещения влево
//! @param keyRight   Код клавиши для смещения вправо
//! @param keyUp      Код клавиши для смещения вверх
//! @param keyDown    Код клавиши для смещения вниз
//!
//! @todo
//! -# Заменить целочисленные координаты на действительные числа (int -> double)
//---------------------------------------------------------------------------------------

void ControlBall (int* vx, int* vy, int keyLeft, int keyRight, int keyUp, int keyDown)
    {
    if (txGetAsyncKeyState (keyLeft))  (*vx)--;
    if (txGetAsyncKeyState (keyRight)) (*vx)++;
    if (txGetAsyncKeyState (keyUp))    (*vy)--;
    if (txGetAsyncKeyState (keyDown))  (*vy)++;
    }

//---------------------------------------------------------------------------------------
//! Определяет столкновения шариков
//!
//! @param x1  x-координата 1-го шарика
//! @param y1  y-координата 1-го шарика
//! @param x2  x-координата 2-го шарика
//! @param y2  y-координата 2-го шарика
//!
//! @note Столкновение происходит, если расстояние между центрами шариков меньше суммы
//!       их радиусов (20).
//!
//! @todo
//! -# Переместить вычисление расстояния в отдельную функцию Distance()
//! -# Выразить магическое число 20 (сумма радиусов шариков) через константы радиусов шариков
//---------------------------------------------------------------------------------------

bool CollideBalls (double x1, double y1, double x2, double y2)
    {
    double distance = sqrt ((x1-x2)*(x1-x2) + (y1-y2)*(y1-y2));
    return (distance < 20);
    }

//---------------------------------------------------------------------------------------
//! Рисует шарик
//!
//! @param x          x-координата шарика
//! @param y          y-координата шарика
//! @param color      Цвет контура шарика
//! @param fillColor  Цвет заполнения шарика
//!
//! @note Радиус шарика = 10.
//!
//! @todo
//! -# Заменить магическое число 20 (радиус шарика) на константу
//---------------------------------------------------------------------------------------

void DrawBall (int x, int y, COLORREF color, COLORREF fillColor)
    {
    txSetColor (color);
    txSetFillColor (fillColor);

    txCircle (x, y, 10);
    }
```

Можно заметить, что вместо функции вычисления расстояния Distance написана немного другая функция, определяющая не расстояние между шариками, а то, столкнулись они или нет. Соответственно, и называется она по-другому: CollideBalls. Дело в том, что вопрос столкновения непростой, и здесь приведен только простейший способ его определения. При высоких скоростях, когда единичное перемещение одного шарика больше диаметра другого шарика, этот метод будет работать не всегда. (Попробуйте решить эту задачу поточнее, но это потребует некоторых усилий в геометрии или в работе с векторами.) Поэтому это скорее заготовка для хорошей функции, но для начала простейший вариант тоже пойдет.

В этой функции выражение “return (distance < 20)” может несколько сбить с толку. Эквивалент этой записи такой:

```
    if (distance < 20)  // Если расстояние меньше 20, то
        return true;    //   верни результат, что столкновение - истина
    else                // иначе
        return false;   //   верни результат, что столкновение - ложь
```

Это весьма понятно, но 1) довольно многословно; 2) можно перепутать return true и return false и получить все наоборот; 3) можно забыть часть “else return false” и получить функцию, возвращающую случайный результат, то есть с неопределенным поведением. Также можно заметить, что значение выражения “distance < 20” совпадает с тем значением, которое мы возвращаем: если оно истинно, то return true, если ложно – return false. Поэтому тот же смысл можно записать так:

```
    bool bangBangCrash = (distance < 20);  // Логическая (булевская) переменная, зависящая от
                                           //   выражения distance < 20
    return bangBangCrash;                  // Верни истинный или ложный результат, записанный
                                           //   в этой переменной
```

Формально, здесь вначале вычисляется значение логической (булевской) переменной bangBangCrash на основе логического выражения “distance < 20”, соответственно, оно может быть либо true (истина), либо false (ложь). Неформально, это можно прочитать как “тот факт, что расстояние меньше 20” (факт истинный или ложный). Это значение затем возвращается как результат функции.

Если это рассуждение понятно, то конструкцию можно сократить до

```
    return /* тот факт, что */ (distance < 20);
```

А, привыкнув совсем, комментарий `/* тот факт, что */` можно убрать.

Если это не показалось удобным, нет проблем, вполне можно писать развернуто. В программировании часто краткость – сестра не таланта, а тонны неясностей и двух тонн рефакторинга впоследствии.

Если разобрались с кодом выше, то вот вторая версия, обладающая хоть какой-то играбельностью: по крайней мере, в нее можно выиграть или проиграть.

```
//---------------------------------------------------------------------------------------------

#include "TXLib.h"

//---------------------------------------------------------------------------------------------

void PlayBall();

void DrawBall (double x, double y, bool active, COLORREF color, COLORREF fillColor);
void MoveBall (double* x, double* y, double* vx, double* vy, double dt);
void ControlBall (double* vx, double* vy, int keyLeft, int keyRight, int keyUp, int keyDown);
bool CollideBalls (double x1, double y1, double x2, double y2);
void Swap (bool* active1, bool* active2);

//---------------------------------------------------------------------------------------------

int main()
    {
    txCreateWindow (800, 600);
    txBegin();

    PlayBall();

    txEnd();
    }

//---------------------------------------------------------------------------------------------

void PlayBall()
    {
    double x1 = 100, y1 = 100, vx1 =  7, vy1 = 5; bool active1 = true;
    double x2 = 100, y2 = 100, vx2 = -7, vy2 = 5; bool active2 = false;
    double dt = 1;

    HDC black = txCreateCompatibleDC (800, 600);

    for (double t = 0; t <= 10000; t += dt)  // Оператор цикла – аналог while. Узнайте про него
        {
        txAlphaBlend (0, 0, black, 0, 0, 0.25);

        printf ("Time LEFT... %05g YCJIOBHbIX secund... Time RIGHT: Unknown! \r", 10000 - t);

        if (CollideBalls (x1, y1, x2, y2) == 1)
            {
            txSetFillColor (TX_RED);
            txClear();

            Swap (&active1, &active2);
            }

        DrawBall (x1, y1, active1, TX_ORANGE, TX_YELLOW);
        DrawBall (x2, y2, active2, TX_GREEN,  TX_LIGHTGREEN);

        MoveBall (&x1, &y1, &vx1, &vy1, dt);
        MoveBall (&x2, &y2, &vx2, &vy2, dt);

        ControlBall (&vx1, &vy1, VK_LEFT, VK_RIGHT, VK_UP, VK_DOWN);
        ControlBall (&vx2, &vy2, 'A',     'D',      'W',    'S');

        txSleep();

        if (txGetAsyncKeyState (VK_ESCAPE))
            {
            if (txMessageBox ("ТОЧНО ДА???",  "Стоп игра", MB_YESNO | MB_ICONQUESTION)    == IDYES &&
                txMessageBox ("УВЕРЕН ДА???", "Стоп игра", MB_YESNO | MB_ICONSTOP)        == IDYES &&
```

```
                txMessageBox ("эх... да?..",  "Стоп игра", MB_YESNO | MB_ICONEXCLAMATION) == IDYES)
                    {
                    txMessageBox ("Ну и ладно. Другие поиграют.", "Жак Фреско");
                    break;
                    }
            }
        }

    printf ("\n\n" "%s is the WINNER!!!\n\n", (active1)? "YELLOW" : "GREEN");

    printf ("Игра закончена, пора рефакторить прогу!\n");

    txDeleteDC (black);
    }

//---------------------------------------------------------------------------------------------

void MoveBall (double* x, double* y, double* vx, double* vy, double dt)
    {
    *x += *vx * dt;
    *y += *vy * dt;

    if (*x >= 800) { *vx *= -0.5; *x = 800; }
    if (*x <=   0) { *vx *= -0.5; *x =   0; }
    if (*y >= 600) { *vy *= -0.5; *y = 600; }
    if (*y <=   0) { *vy *= -0.5; *y =   0; }

    *vx *= 0.995;
    *vy *= 0.995;
    }

//---------------------------------------------------------------------------------------------

void ControlBall (double* vx, double* vy, int keyLeft, int keyRight, int keyUp, int keyDown)
    {
    double dv = 0.25;

    if (txGetAsyncKeyState (keyLeft))  *vx -= dv;
    if (txGetAsyncKeyState (keyRight)) *vx += dv;
    if (txGetAsyncKeyState (keyUp))    *vy -= dv;
    if (txGetAsyncKeyState (keyDown))  *vy += dv;

    if (txGetAsyncKeyState (VK_SPACE)) { *vx /= 2; *vy /= 2; }
    }

//---------------------------------------------------------------------------------------------

bool CollideBalls (double x1, double y1, double x2, double y2)
    {
    double distance = sqrt ((x1-x2)*(x1-x2) + (y1-y2)*(y1-y2));

    return (distance < 50);
    }

//---------------------------------------------------------------------------------------------

void DrawBall (double x, double y, bool active, COLORREF color, COLORREF fillColor)
    {
    txSetColor     (color, 1);
    txSetFillColor (fillColor);

    txCircle (x, y, 25);

    if (active)
        {
```

```
        txSetFillColor (TX_WHITE);
        txCircle (x, y, 15);
        }
    }

//---------------------------------------------------------------------------------------------

void Swap (bool* active1, bool* active2)
    {
    bool temp = *active1;
    *active1 = *active2;
    *active2 = temp;
    }
```

Во-первых, отметим, что этот код без документации. Это очевидный минус. Он более-менее понятен потому, что перед ним была приведена предыдущая версия с документацией и пояснениями в тексте. Если бы этого не было, приходилось бы "цепляться" за имена функций, параметров и переменных, а если бы и это было на низком уровне, то вряд ли мы бы чего-нибудь поняли. Умение читать код – ключевое для программиста и занимает гораздо большую часть рабочего времени, чем написание кода, кстати. Но если у вас не читается плохой, неясный код, плохо разделенный на функции и с плохими названиями – это определенно не ваша вина.

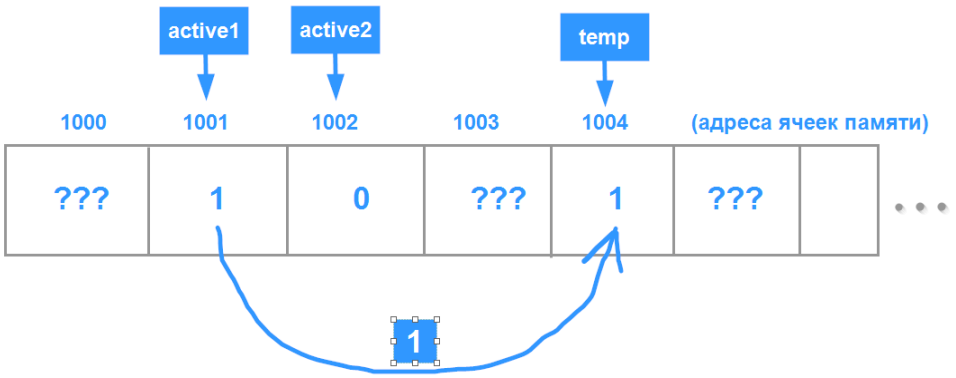
Во второй версии добавлены вот какие изменения (погодите пока это читать, сравните версии и разберите код выше сами):

Во-первых, у шариков появилась "активность", активный шарик содержит яркую белую часть внутри. Ее рисует, конечно, функция DrawBall, теперь у нее есть параметр активности, если он равен true, то поверх шарика дорисовывается белый круг.

В функции PlayBall появились переменные, соответствующие активностям шариков. Если шарики сталкиваются, то значения переменных меняются местами с помощью функции Swap. В нее передаются, соответственно, указатели (адреса) переменных active1 и active2, чтобы функция Swap смогла "залезть" по этим адресам и обменять их значения. Если не очень понятно, как три присваивания внутри Swap меняют значения переменных, представьте себе, что вместо функции Swap эти присваивания написаны прямо в функции PlayBall:

```
    if (CollideBalls (x1, y1, x2, y2) == 1)
        {
        ...
        bool temp = active1;  // Действие 1
        active1 = active2;    // Действие 2
        active2 = temp;       // Действие 3
        }
```

И нарисуйте вот такой рисунок, на котором изображены переменные active1, active2 и временная переменная temp после совершения первого действия:



Нарисуйте стрелочки, ведущие от одной переменной к другой, которые покажут, как копируются значения при присваиваниях, и пронумеруйте их соответственно действиям. После этого на примере конкретных значений разберите, как и почему происходит обмен.

Также в PlayBall добавлен довольно кустарный отсчет времени, по истечении которого тот шарик, который сохранил активность, выигрывает.

```
    printf ("Time LEFT... %05g YCJIOBHbIX secund... Time RIGHT: Unknown! \r", 10000 - t);
```

В конце цикла при нажатии клавиши Escape срабатывает условие, позволяющее досрочно выйти из игры.

В функции MoveBall добавлен эффект силы трения:

```
    *vx *= 0.995;
    *vy *= 0.995;
```

В функции управления также добавлен чит-код.

Как еще можно улучшить эту игру?

Предположим, мы хотим сделать области, где поведение объектов меняется. Это могут быть области ускорения или замедления, изменения цвета объектов, или еще каких-либо эффектов. Как определить, что шарик пролетает по такой области? Проще всего, если область более-менее простой формы, например, круг, прямоугольник, овал, или другой простой объект, или их объединение или пересечение. Тогда можно написать неравенство или систему неравенств, определяющих, лежит ли точка в такой области. Такую систему надо записать в функцию с ясным названием, принимающую координаты шарика (x, y), и возвращающую логическое значение, показывающее истинность неравенства или системы в этих координатах. Это будет напоминать CollideBalls, но с двумя параметрами. Чем сложнее неравенства, тем выгоднее будет поместить их в отдельной функции. Кстати, в ЕГЭ есть или было задание на такую тему.

Другой способ определить попадание в такую зону можно примерить, если эти ее нарисовать определенным цветом, например, красным, то есть чтобы зона постоянно была видна на экране. Тогда можно воспользоваться функцией txGetPixel, она возвращает цвет точки экрана с заданными координатами. Если этот цвет – красный, то есть он совпадает с цветом области, то шарик находится внутри нее и его характеристики и поведение меняются. (Заведите для такого цвета константу, чтобы не вписывать его напрямую и легко менять, если что.) Если хочется, чтобы область была невидимой, можно пойти на такой трюк: нарисуйте ее цветом, близким к

цвету фона, например, если фон черный (это TX_BLACK или RGB (0, 0, 0)), то цвет области будет RGB (0, 0, 1) - это очень темно-синий. Небольшое отличие, когда какая-то цветовая компонента отличается всего на единицу, человеческий глаз не увидит, а условный оператор, точно сравнивающией цвета, заметит. (Вот видите, цвет уже стал не красный, а темно-синий, но так как у вас есть константа, ее значение легко изменить.)

Таких зон-ловушек можно сделать несколько и для тренировки задать их разными способами. Они, кстати, могут быть и не только ловушками: это могут быть порталы или переход на следующий уровень.

2.6. Структуры

Возможно, вы заметили, что развитием программы начали добавляться переменные, имеющие отношения к шарикам. Промасштабируем это: в будущем шарики можно сделать разных радиусов, меняющихся цветов, добавить еще какие-либо свойства, и все это потребует новых переменных, которые придется добавлять для каждого шарика. Все это довольно неприятно, потому что если шариков несколько, то каждую переменную (например, active) нужно добавлять в каждый шарик вручную:

```
    double x1 = 100, y1 = 100, vx1 =  7, vy1 = 5; bool active1 = true;
    double x2 = 100, y2 = 100, vx2 = -7, vy2 = 5; bool active2 = false;
```

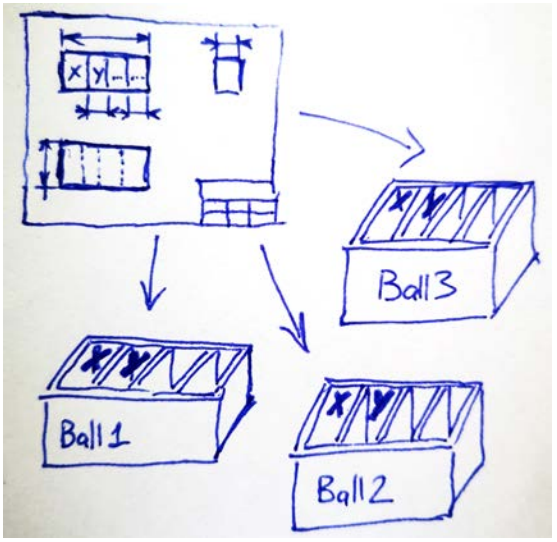
Язык программирования пока не помогает нам сделать так, чтобы нужная переменная сама автоматически “добавилась” во все шарики.

Это потому, что, собственно, в нашей программе нет явного понятия шарика. Неявное понятие есть: совокупность переменных, они даже объявлены на одной строке, чтобы подчеркнуть их связь. Но это понимает человек, пишущий и читающий программу (и не всегда это сразу понятно при чтении), но с точки зрения языка программирования явно это не выражено и поэтому не особо нам помогает. Следить за связью переменных тоже приходится вручную, что может привести к таким, например, ошибкам:

```
    MoveBall (&x1, &y1, &vx2, &vy2, dt);
    MoveBall (&y2, &x2, &vx1, &vy2, dt);
```

В первой строке вместо скоростей первого шарика взяты скорости второго, а во второй строке перепутаны координаты x и y, а скорость vx взята из второго шарика. Такие ошибки трудно находить, кстати. И компилятор не помогает, потому что с его точки зрения все эти переменные независимы друг от друга, не образуют какую-то смысловую группу, не описывают какой-то единый объект.

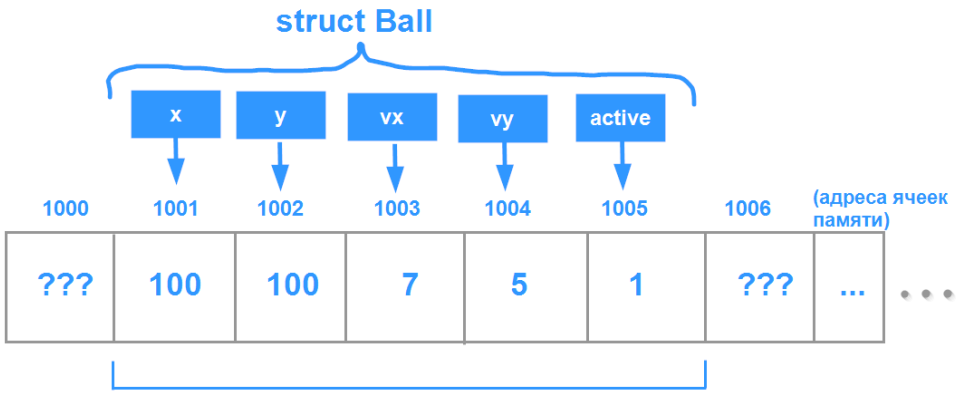
К счастью, в языке Си есть понятие структуры, описывающей группу переменных, имеющих общий смысл. Если сначала описать, из каких переменных состоит такая группа (структура), то потом можно легко создавать переменные, содержащие другие переменные, связанные по смыслу друг с другом. Это напоминает создание чертежа ящика с отделениями для переменных и изготовление нескольких ящиков по такому чертежу:



Вот как такой "чертеж" (определение структуры) выглядит в коде:

```
struct Ball          // Определение структуры Ball - группы переменных для описания шарика
    {
    double x, y;     // Координаты шарика
    double vx, vy;   // Скорости шарика
    bool active;     // Активность шарика
    };
```

Вот как это выглядит в оперативной памяти компьютера (схема немного упрощена):



Переменные, перечисленные в структуре, называются полями структуры. Обратите внимание, что они тут также перечислены по подгруппам, каждая подгруппа - на своей строке. Если у вас потом пойдут мысли о создании других групп-структур, полезных для описания координат (с компонентами x и y), скоростей (с компонентами x и y) и ускорений (с теми же компонентами... хм, это же вектор...), то это правильные мысли. Но давайте сейчас разберемся с простым примером.

Поскольку он повлияет на всю программу, разместить это определение нужно до прототипов функций, после инклюдов и констант.

Теперь можно легко создавать структурные переменные:

```
    Ball ball1 = { 100, 100,  7, 5, true  };
    Ball ball2 = { 100, 100, -7, 5 };

//  ball1
```

```
//    \   x     y     vx   vy  active
// ... +-----+-----+-----+-----+-----+ ...
//     | 100 | 100 |  7  |  5  |  1  |
// ... +-----+-----+-----+-----+-----+ ...

//  ball2
//    \   x     y     vx   vy  active
// ... +-----+-----+-----+-----+-----+ ...
//     | 100 | 100 | -7  |  5  |  0  |
// ... +-----+-----+-----+-----+-----+ ...
```

или, с именами полей структуры,

```
    Ball ball1 = { .x = 100, .y = 100, .vx =  7, .vy = 5, .active = true  };
    Ball ball2 = { .x = 100, .y = 100, .vx = -7, /vy = 5 };
```

Поля, не указанные в конце перечисления значений (поле active для ball2), устанавливаются равными нулю.

Обращаться к элементам (частям) структуры тоже легко:

```
    ball1.x   // Это x-координата первого шарика
    ball2.vx  // Это скорость по Y второго шарика
```

Но самое главное, что структурные переменные можно передавать как единое целое в функции, не боясь запутаться:

```
    MoveBall (&ball1, dt);
    MoveBall (&ball2, dt);
```

Это круто, но для этого, конечно, надо отрефакторить функцию MoveBall:

```
void MoveBall (Ball* ball, double dt)
    {
    (*ball).x += (*ball).vx * dt;
    (*ball).y += (*ball).vy * dt;
    ...
    }
```

Неуклюжие выражения `(*ball).x` и им подобные возникают снова по причине неудачного сочетания приоритета операторов доступа по указателю (*) и доступа к полю структуры (.). Чтобы не писать так каждый раз, есть специальный оператор -> (оператор "стрелочка", это знак минуса и затем знак "больше" без пробелов между ними), заменяющий и звездочку, и точку, и скобки. С ним получается так:

```
void MoveBall (Ball* ball, double dt)
    {
    ball->x += ball->vx * dt;
    ball->y += ball->vy * dt;
    ...
    }
```

"Звездочки, скобочки, стрелочки в ряд – трамвай нам привез программистов отряд". :)

Зато теперь не проблема сделать не два, а, скажем, три-четыре шарика, с помощью всего трех-четырех переменных, а не двадцати (!).

Некоторые, возможно, захотят по инерции включить в структуру dt, раз уж она была объявлена вместе со старыми переменными шариков. Но смотрите: по факту, dt задает скорость течения времени. Разве время течет по-разному для разных шариков? Ну разве что мы учитываем релятивистские эффекты при скоростях, близких к скорости света. :) Раз в игру это не заложено (а сможете ли вы так быстро нажимать на клавиши?), то не надо и зазря дублировать переменную.

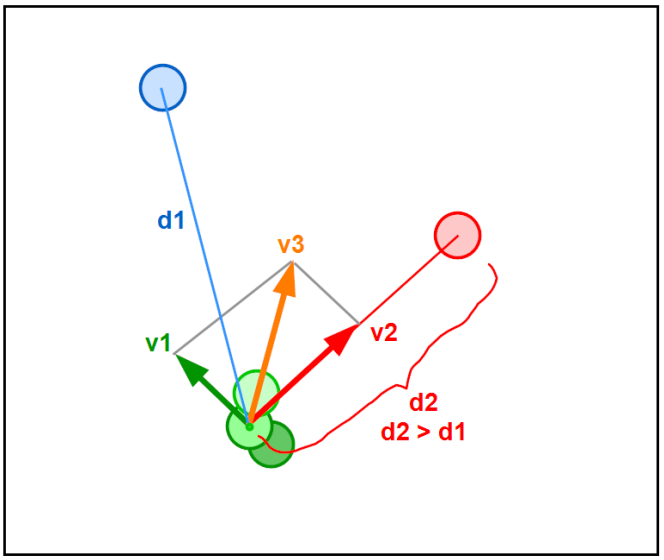
Как видим, с помощью структур мы последовали той же стратегии, что и с функциями – “разделяй и властвуй”. Мы видели, что в коде есть совокупность строк, объединенных общим смыслом, алгоритмом, выделяли ее в отдельную функцию, называли понятным именем и потом легко использовали в коде.

Теперь мы видим совокупность переменных, цельным образом описывающую некоторый объект, и определяем структуру, которая теперь будет отвечать за этот объект. Называем ее понятным всем именем, помещаем туда все переменные объекта. Для того, чтобы было проще работать со структурой, пишем функции для всех действий с ней. И легко используем в коде.

Ближайшим аналогом структур в C++ являются классы, фактически, это то же самое. Сейчас мы не будем лезть в их синтаксис, но позже разберем.

2.6.1. Задание

Вы, наверное, уже поняли. Конечно, нужен рефакторинг с использованием структур. Ну и добавьте, скажем, третий шарик, пусть он гоняется за тем шариком, который ближе к нему, как NPC (неигровой персонаж). Определить расстояния до шариков вы сможете (теорема Пифагора + функция Distance + возвращаемое значение), а дальше подумайте, как заставить двигаться третий шарик в сторону ближайшего, поможет вам в этом физика и векторная алгебра:



Получается, для третьего шарика будет своя функция физики, отличающаяся от уже написанной. Когда будете ее писать, назовите так, чтобы было понятно, для кого она предназначена.

Также можно реализовать игру “Арканоид” или “Теннис” (“Pong”), старейшую из компьютерных игр. В ней будет два вида объектов: мяч (у нас он уже есть) и “ракетка”, отталкивающая его. В дополнение к функциям мяча и ракетки, напишите функцию, оценивающую проигрыш, так как в ней потребуется проверять несколько условий столкновения.

Еще один момент, который наверняка давно не нравится физикам – мы совершенно не учитываем массу объектов. Учет массы делает движение более реалистичным, а игру – интересной. От массы может зависеть управление объектом, так как управляющее воздействие есть приложение к объекту силы, а оно влияет на изменение скорости в зависимости от массы по второму закону Ньютона. На этой основе можно написать, например, старинную, но интересную игру "посадка на планету". С одной из планет стартует космический танкер, полностью наполненный топливом, и поэтому очень тяжелый. Он должен достигнуть другой планеты и успешно сесть на нее. Корабль управляется четырьмя реактивными двигателями, направленными влево, вправо, вверх и вниз (это наши старые знакомые txGetAsyncKeyState (VK_LEFT) и другие). Поначалу управлять им нелегко из-за его массы, и для разгона и торможения требуется много нажатий на управляющие клавиши. Двигатели потребляют то же топливо, что везет танкер, поэтому при их работе масса корабля уменьшается. Через некоторое время управлять становится легче. Однако если корабль делает слишком много маневров, то топливо в танкере заканчивается, и из-за малой массы при каждом управляющем воздействии корабль начинает болтать, как жестянку. Критерием качества посадки считается значение импульса при касании планеты. Если оно достаточно мало, корабль сел успешно, если велико – он разбился. Получается такой тренажер на инстинктивное осознание второго закона Ньютона. Попробуйте, и, может быть, вас возьмут в космическую команду.

2.7. Работа с изображениями

Если в игре хочется красивый фон, то его можно нарисовать, сфотографировать или скачать, сохранить в виде картинки, загрузить в программе и отобразить в программе вместо стирания с экрана. Библиотека TXLib позволяет загружать изображения только в формате BMP, это обусловлено ограничениями системы рисования Windows GDI, на которой основан TXLib. Если вы скачали изображение в другом формате, его можно открыть в растровом графическом редакторе и сохранить в формате BMP. Картинки можно использовать не только для фона, но и для персонажей игры.

Пусть, например, мы хотим сделать игру про одного итальянского сантехника.

Погнали:

```
void PlayMario()
    {
    int x = 100, y = 100, vx = 7, vy = 5;
    ...
    }
```

Погодите.

А если персонажей будет три? Четыре, пять вместе с NPC? Опять 20 переменных? Хммм.

Ах да, структуры.

```
struct Mario        // Пока - только для Марио. Далее будем использовать для других персонажей
    {
    double x, y;    // Координаты
    double vx, vy;  // Скорости
    ...             // Возможно, добавим что-то еще
    };
```

Вот это дело. Теперь можно писать функцию игры:

```
void PlayMario()
    {
    Mario mario = { .x = 100, .y = 500, .vx = 2, .vy = 0 };
    ...
    }
```

Обратите внимание: при задании начальных значений в структуре можно писать имена ее компонентов, с точкой. Так не запутаешься, что к чему относится.

Дальше надо загрузить изображения Марио и фона игры из файлов с картинками в оперативную память компьютера. Делается это простой функцией из библиотеки TXLib (прочитайте о ней в системе помощи):

```
    HDC backImage = txLoadImage ("MarioBack.bmp");
```

Переменная backImage, которую возвращает функция txLoadImage, имеет не очень понятный сразу тип HDC. Он в Windows отвечает за изображения, загруженные в память компьютера, но не обязательно отображенные на экране. Грубо говоря, это некоторый номер, который Windows присваивает изображению при его загрузке.

Имейте в виду, что **изображения поддерживаются только в формате BMP.** Если взять файл другого формата, например JPG, и переименовать его со сменой расширения на BMP, то от этого формат не изменится. Такое изображение загружено не будет.

Если изображение не загрузилось и функция txLoadImage вернула NULL, то надо прежде всего **проверить наличие файла изображения по указанному в программе пути** и формат файла. Если путь к файлу не указан (или указан как неполный), то путь отсчитывается от текущей папки программы, которая может не совпадать текущей папкой среды программирования. Текущую папку программы можно посмотреть по команде About в системном меню (она указана там как "Run from").

Если файл по-прежнему не загружается, **откройте его в редакторе Microsoft Paint** (нажмите `<WIN+R> mspaint <ENTER>`) или другой программе обработки изображений и **сохраните его заново как изображение в формате BMP.** Это иногда бывает нужно для файлов, преобразованных из других форматов программами-конверторами или сайтами для преобразования изображений.

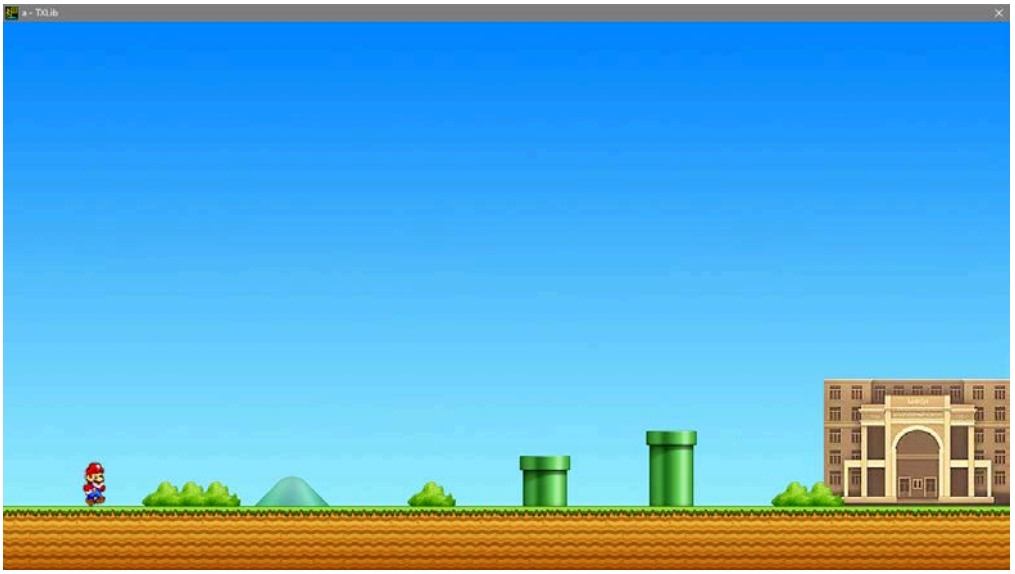
Загруженные изображения можно копировать на экран функциями txBitBlt, txTransparentBlt и txAlphaBlend. Прочитайте о них в системе помощи. Самая простая – txBitBlt, она копирует изображение без учета прозрачности. Остальные две разными способами позволяют задать прозрачные участки, это полезно при копировании персонажей.

```
    txBitBlt (0, 0, backImage);                  // Копирование фоновой картинки на экран
    ...
    txAlphaBlend (mario.x, mario.y, marioImage);  // Копирование картинки с Марио на экран
```

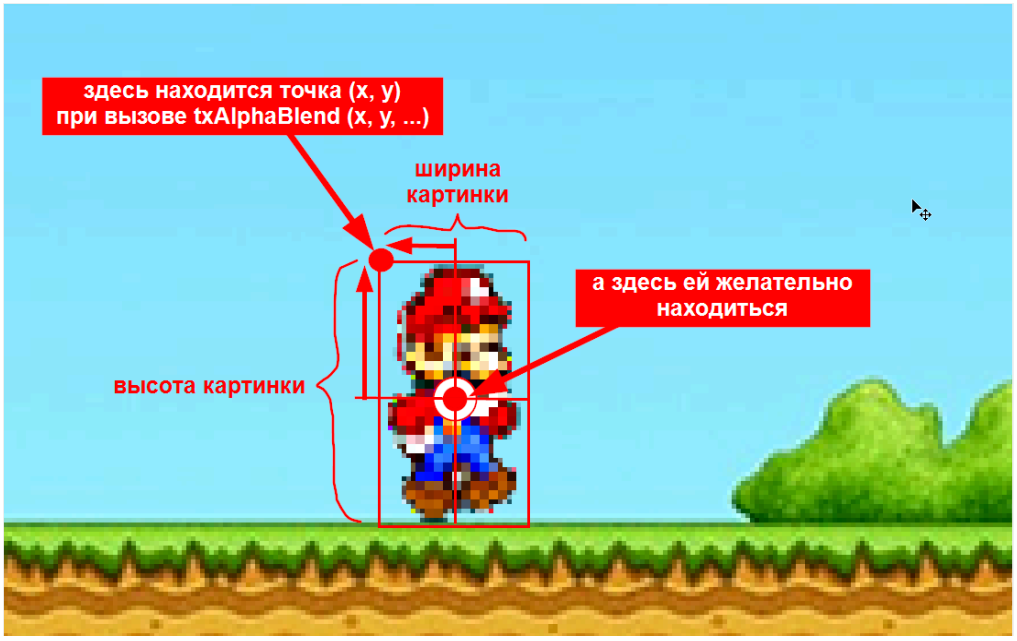
Когда изображения, загруженные при помощи txLoadImage, больше не нужны, их надо обязательно удалить из оперативной памяти функцией txDeleteDC. Если этого не сделать, возможно, вскорости придется перезагружать Windows, так как операционная система не может загружать слишком много изображений одновременно.

```
    txDeleteDC (backImage);
    txDeleteDC (marioImage);
```

Вот что получается:



С выводом изображения Марио есть нюанс. Функции копирования картинок принимают координаты не центра изображения, а его левого верхнего угла. Поэтому при движении Марио может "погружаться" под землю. Лучше выводить картинку с учетом этого, смещая изображение на половину его размера влево и вверх, чтобы координаты x и y в структуре все-таки соответствовали центру картинки.



Теперь давайте промасштабируем наше решение по движущимся объектам. В игре может двигаться не только главный персонаж. Могут двигаться второстепенные персонажи, детали фона – облака, например. Могут появляться и исчезать препятствия. Что нужно для описания всего этого? Структура для описания движущегося объекта у нас уже есть, что стоит добавить? Надо добавить то, что необходимо для их функций – как минимум, рисования и движения, и добавить то, что будет у разных объектов различным.

Так как объекты имеют размеры, стоит добавить эти размеры, например, sizeX и sizeY, или width и height. Если объекты будут рисоваться с помощью копирования изображений, то стоит добавить переменную, отвечающую за изображение, HDC image, и заодно имя файла, откуда брать это изображение. Также объекты будут по-разному реагировать на притяжение Земли –

облака, например, не будут падать (возможно). Поэтому добавим ускорение, для общности сразу два, ax и ay.

С движением разных объектов тоже будут нюансы – законы движения могут быть разными. Например, физика бегающего шарика для Марио уже не очень годится, ведь он не будет отражаться от земли. Вместо этого должна обнуляться его скорость. Облака не подчиняются силе трения, и после того, как скроются за одним краем экрана, должны появляться из-за другого (сделать это очень просто: если облако плывет слева направо и скрывается за правым краем экрана, присвойте ему отрицательную координату, сравнимую с его шириной. Оно выплывет из левого края экрана). То есть функции движения будут разные, но все они будут принимать структуру объекта как параметр.

Тем не менее, в движении разных объектов будут общие мотивы: перемещение будет определяться скоростью, изменение скорости – ускорением. Чтобы это не дублировать во всех функциях движения (и иметь проблемы с их изменениями), лучше сделать отдельную функцию "базовой физики" и вызывать ее в начале всех функция движения.

Получается вот что:

```
//-----------------------------------------------------------------------------------------
//! Структура с информацией об объекте
//-----------------------------------------------------------------------------------------

struct GameObject
    {
    const char* imageFile;   //!< Имя файла с картинкой. Тип const char* - это текстовая строка.
    double x,  y;            //!< Координаты объекта
    double vx, vy;           //!< Скорости
    double ax, ay;           //!< Ускорения

    bool active;             //!< Неактивные объекты не рисуются, см. DrawGameObject

    HDC image;               //!< Переменная для хранения HDC изображения
    int sizeX, sizeY;        //!< Размеры изображения
    };

//-----------------------------------------------------------------------------------------
//! Рисование игрового объекта
//!
//! @param object  Структура с информацией об объекте
//!
//! @note Для отладки, если нажата клавиша Caps Lock, сначала на месте объекта рисуется
//!       розовый кружок. Если картинка почему-то не отобразилась, то кружок будет виден.
//!
//! @todo Параметр object будет каждый раз копироваться при вызове функции, это безопасно,
//!       но дорого, надо с этим что-то сделать...
//-----------------------------------------------------------------------------------------

void DrawGameObject (GameObject object)
    {
    if (GetKeyState (VK_CAPITAL))      // Если включен режим Caps Lock. Можно заменить на
        {                              // VK_SCROLL, это клавиша Scroll Lock, она редко
        txSetColor (TX_RED);           // используется в программах и удобна для отладки
        txSetFillColor (TX_PINK);
        txCircle (object.x, object.y, 5);
        }
```

```
    if (!object.active)                 // Помните что если return выполняется, то функция
        return;                         // сразу завершает работу? Тогда картинка не нарисуется.
    txAlphaBlend (object.x - object.sizeX/2, object.y - object.sizeY/2, object.image);
    }

//-----------------------------------------------------------------------------------------
//! Перемещение объекта по базовым законам физики
//!
//! @param object  Структура с информацией об объекте
//! @param dt      Интервал времени движения
//!
//! @note Отталкивания нет, только равномерное/равноускоренное движение
//-----------------------------------------------------------------------------------------

void BaseMove (GameObject* object, double dt)
    {
    object->vx += object->ax * dt;
    object->vy += object->ay * dt;

    object->x  += object->vx * dt;
    object->y  += object->vy * dt;
    }

//-----------------------------------------------------------------------------------------
//! Перемещение Марио
//!
//! @param mario     Структура с информацией об объекте
//! @param dt        Интервал времени движения
//! @param scrSizeX  Размер области движения по X
//! @param scrSizeY  Размер области движения по Y
//!
//! @note
//!   - От верхней, правой и левой стенок отталкивание зеркальное
//!   - От нижней отталкивания нет, падение останавливается (vy = 0)
//!
//! @todo
//! -# Сделать отталкивание от стенок с учетом размера Марио
//! -# Убрать возможность отталкиваться от воздуха в прыжке
//! -# Коэффициент трения 0.9 - в константу!
//-----------------------------------------------------------------------------------------

void MoveMario (GameObject* mario, double dt, int scrSizeX, int scrSizeY)
    {
    BaseMove (mario, dt);                       // Расчет базовой физики

    if (mario->x > scrSizeX)
        {
        mario->vx = - mario->vx;
        mario->x  = 2 * scrSizeX - mario->x;
        }

    if (mario->y > scrSizeY)
        {
        mario->vy = 0;                          // Столкновение с землей, прекращаем падение
        mario->y  = scrSizeY;
        }

    if (mario->x < 0)
        {
        mario->vx = - mario->vx;
        mario->x  = - mario->x;
        }

    if (mario->y < 0)
        {
```

```
        mario->vy = - mario->vy;
        mario->y  = - mario->y;
        }

    mario->vx *= 0.9;
    mario->vy *= 0.9;
    }
```

Получается что-то вроде библиотеки для работы и игровыми объектами. Только она не вынесена в отдельный файл. (А вы возьмите и вынесите.)

Вот как могла бы выглядеть функция игры, если мы добавим другие движущиеся объекты, например, облака:

```
void PlayMario()
    {
    HDC backImage = txLoadImage ("MarioBack2.bmp");
    if (!backImage) { txMessageBox ("Не могу загрузить 'MarioBack.bmp'"); return; }

    GameObject mario = { .imageFile = "MarioSprite.bmp",
                         .x = 100, .y = 100,
                         .vx = 2, .vy = 0, .ax = 0, .ay = 1,
                         .active = true };

    mario.image = txLoadImage (mario.imageFile);
    if (!mario.image) { txMessageBox ("Не могу загрузить картинку Марио"); return; }
    mario.sizeX = txGetExtentX (mario.image);  // Посмотрите, что делают эти функции,
    mario.sizeY = txGetExtentY (mario.image);  // в системе помощи TXLib
    txUseAlpha (mario.image);

    GameObject clouds1 = { .imageFile = "MarioClouds1.bmp",
                         .x = 100, .y = 100, .vx = 1, .vy = 0, .ax = 0, .ay = 0,
                         .active = true };

    GameObject clouds2 = { .imageFile = "MarioClouds2.bmp",
                         .x = 1000, .y = 120, .vx = -1, .vy = 0, .ax = 0, .ay = 0,
                         .active = true };

    clouds1.image = txLoadImage (clouds1.imageFile);
    if (!clouds1.image) { txMessageBox ("Не могу загрузить картинку облаков 1"); return; }
    clouds1.sizeX = txGetExtentX (clouds1.image);
    clouds1.sizeY = txGetExtentY (clouds1.image);
    txUseAlpha (clouds1.image);

    clouds2.image = txLoadImage (clouds2.imageFile);
    if (!clouds2.image) { txMessageBox ("Не могу загрузить картинку облаков 2"); return; }
    clouds2.sizeX = txGetExtentX (clouds2.image);
    clouds2.sizeY = txGetExtentY (clouds2.image);
    txUseAlpha (clouds2.image);

    int scrSizeX = txGetExtentX();
    double dt = 1;

    while (! GetAsyncKeyState (VK_ESCAPE))
        {
        txBitBlt (0, 0, backImage);

        DrawGameObject (mario);
        DrawGameObject (clouds1);
        DrawGameObject (clouds2);

        MoveMario  (&mario, dt, scrSizeX, 590); //!< @todo Y-коорд. земли на картинке - в константу
        MoveClouds (&clouds1, dt, scrSizeX);
```

```
        MoveClouds (&clouds2, dt, scrSizeX);

        ControlMario (&mario, VK_LEFT, VK_RIGHT, VK_UP, VK_DOWN, VK_SPACE);

        txSleep();
        }

    txDeleteDC (backImage);
    txDeleteDC (mario.image);
    txDeleteDC (clouds1.image);
    txDeleteDC (clouds2.image);
    }
```

Ну как вам сказать. Главный цикл игры (while) компактен и прост, это хорошо. Но очень смущает загрузка картинок, она там скопирована три раза, потому что загружаются три картинки движущихся объектов. Такое себе. Надо что-то с этим сделать (и сделать, очевидно, функцию загрузки). Вообще, вот вам универсальный способ решения проблем:

1. Есть проблема
2. Выделяем проблему в функцию
3. Нет проблемы
4. PROFIT!!!

Как-то так. :)

Пишем функцию:

```
void LoadGameImage()
    {
    GameObject object = { .imageFile = "MarioClouds1.bmp",
                          .x = 100, .y = 100, .vx = 1, .vy = 0, .ax = 0, .ay = 0,
                          .active = true };

    object.image = txLoadImage (object.imageFile);
    if (!object.image) { txMessageBox ("Не могу загрузить картинку облаков 1"); return; }
    object.sizeX = txGetExtentX (clouds1.image);
    object.sizeY = txGetExtentY (clouds1.image);
    txUseAlpha (object.image);
    }
```

Так, посмотрите на этот код.

Не читайте дальше. Критически посмотрите на этот код.

Погодите. Это же просто напрямую вынесенный кусок из основной программы, с переименованным clouds1 в object, причем даже не во всех местах. Оно даже компилироваться не будет. Ну хорошо, допереименуем, скомпилируется. И что? Мы же не сможем использовать переменную object снаружи функции. Дальше, зачем здесь задавать положение объекта и его скорости? Хорошая функция решает одну ясную задачу, а тут каша из смыслов. Ну и напоследок – загружаться будет всегда один и тот же файл, MarioClouds1.bmp. Тут тоже не изменили ничего.

Нет, такое нам не надо. Функция загрузки должна делать хорошо то, что надо, а что не надо – не делать вообще. Она должна принимать параметры, делающие ее гибкой. Она должна позволять использовать структурную переменную игрового объекта после возврата из себя. Давайте подумаем о входе и выходе функции – об ее интерфейсе.

Еще вот что важное: давайте пока не будем детально думать, как мы напишем нашу функцию внутри нее. Поиграем с ее вызовом, поищем удобный вариант, так, как будто эта функция уже написана. В конце концов, вызываем мы функцию много чаще, чем ее определяем. И пользуемся, уже зачастую забыв, как в точности она устроена внутри – это называется "черным ящиком". Поэтому удобство вызова важнее деталей реализации внутри функции. (Если только мы не захотим такой способ вызова, который приведет к долгим и неэффективным действиям внутри, тогда придется идти на компромисс, сохраняя эффективность функции.) Такой подход называется проектированием сверху вниз.

При вызове функции мы будем передавать через параметры все, что должно лежать в структуре.

Функция будет создавать внутри себя структурную переменную игрового объекта, записывать в нее все параметры, загружать картинки и определять их размеры.

При выходе из функции она будет возвращать значение этой структурной переменной. Во время возврата значения компилятор скопирует или переместит все поля структуры в соответствующую переменную вызывающей функции (PlayMario). Эта переменная сохранит возвращенное значение и позволит его использовать:

```
void PlayMario()
    {
    ...
    GameObject mario   = LoadGameObject ("MarioSprite.bmp",   100, 100,  2, 0, 0, 1, true);
    GameObject clouds1 = LoadGameObject ("MarioClouds1.bmp",  100, 100,  1, 0, 0, 0, true);
    GameObject clouds2 = LoadGameObject ("MarioClouds1.bmp", 1000, 100, -1, 0, 0, 0, true);
    ...
    DrawGameObject (mario);
    DrawGameObject (clouds1);
    DrawGameObject (clouds2);
    ...
    }
```

Вообще у нас получилась скорее не загрузка картинки, а вообще "начальная установка" объекта – его инициализация, или создание. Ок, переименуем:

```
void PlayMario()
    {
    ...
    GameObject mario   = CreateGameObject ("MarioSprite.bmp",   100, 100,  2, 0, 0, 1, true);
    GameObject clouds1 = CreateGameObject ("MarioClouds1.bmp",  100, 100,  1, 0, 0, 0, true);
    GameObject clouds2 = CreateGameObject ("MarioClouds1.bmp", 1000, 100, -1, 0, 0, 0, true);
    ...
    }
```

Стало яснее. Если использовать параметры по умолчанию (ax = 0, ay = 0, active = true) то получится короче:

```
void PlayMario()
    {
    ...
    GameObject mario   = CreateGameObject ("MarioSprite.bmp",  100,  100,  2, 0, 0, 1);
    GameObject clouds1 = CreateGameObject ("MarioClouds1.bmp", 100,  100,  1, 0);
    GameObject clouds2 = CreateGameObject ("MarioClouds2.bmp", 1000, 100, -1, 0);
    ...
    }
```

Уже лучше.

Тут, правда, потерялись имена компонентов структур, которые мы раньше могли указывать при инициализации структурных переменных. Если это снижает понятность кода, то можно сделать и по-другому. Если не снижает, то все равно разберите другой вариант ниже. Он может пригодиться в других случаях, когда не происходит создания объекта, а что-то меняется в середине его жизни (например, загружаются новые картинки при переходе на другой уровень).

Сама функция CreateGameObject могла бы выглядеть так:

```
GameObject CreateGameObject (const char* imageFile,
                             double x, double y, double vx, double vy, double ax, double ay,
                             bool active)
    {
    GameObject object = { imageFile, x, y, vx, vy, ax, ay, active };  // Создаем игровой объект

    object.image = txLoadImage (object.imageFile);
    if (!object.image) { txMessageBox ("Не загрузилась картинка"); return object; }

    object.sizeX = txGetExtentX (clouds1.image);
    object.sizeY = txGetExtentY (clouds1.image);
    ...

    return object;  // Возвращаем объект
    }
```

Параметр imageFile типа const char* – это строка символов, содержащая имя файла с картинкой. (Более строго – адрес того места в памяти, где лежит эта строка символов. Чтобы функция случайно не испортила это имя, перед типом char* стоит ключевое слово const.) При вызове оно задается текстом в двойных кавычках.

Получается, что переменные mario, clouds1 и clouds2 создаются и получают свои значения в результате работы функции CreateGameObject. Поэтому игровые объекты в функцию не передаются, а создаются внутри нее и возвращаются через возвращаемое значение, попадая в нужную переменную на вызывающей стороне.

Теперь разберем другой вариант функции. В нем мы сделаем по-другому: пусть структурная переменная mario, clouds1 или clouds2 будет создана в PlayMario до вызова нашей функции, как это было изначально в самом первом примере со структурами. При создании (инициализации) мы запишем в нее все нужные нам значения, определяющие игровой объект – имя файла с картинкой, координаты, скорости и т.п. Некоторые значения – HDC картинки и ее размеры – мы опустим. Тогда наша функция может принять эту структуру, взять из нее имя файла, загрузить из него картинку и записать HDC картинки в структуру. Далее функция определит размеры картинки и тоже запишет в структуру, и т.д. В ходе работы функции структура будет изменяться, поэтому надо передать эту структуру по указателю.

То есть, в отличие от первого варианта, функция не будет создавать внутри себя никакого игрового объекта. Она примет его в форме адреса от вызывающей функции PlayMario, и изменит его содержимое, созданное и по-прежнему принадлежащее функции PlayMario.

Вот этот вариант в коде. Назовем нашу функцию LoadGameObject, чтобы не путать с первым вариантом. Сначала вызов, который всему голова:

```
void PlayMario()
    {
```

```
    ...
    GameObject mario   = { "MarioSprite.bmp",  100, 100,  2, 0, 0, 1, true };
    LoadGameObject (&mario);

    GameObject clouds1 = {"MarioClouds1.bmp",  100, 100,  1, 0, 0, 0, true };
    LoadGameObject (&clouds1);

    GameObject clouds2 = {"MarioClouds2.bmp", 1000, 100, -1, 0, 0, 0, true };
    LoadGameObject (&clouds1);
    ...
    }
```

или так:

```
void PlayMario()
    {
    ...
    GameObject mario   = { "MarioSprite.bmp",  100, 100,  2, 0, 0, 1, true };
    GameObject clouds1 = {"MarioClouds1.bmp",  100, 100,  1, 0, 0, 0, true };
    GameObject clouds2 = {"MarioClouds2.bmp", 1000, 100, -1, 0, 0, 0, true };

    if (!LoadGameObject (&mario))   return;  // Марио не загрузился - завершаем игру
    if (!LoadGameObject (&clouds1)) return;  // Облака 1 не загрузились - завершаем игру
    if (!LoadGameObject (&clouds2)) return;  // Облака 2 не загрузились - завершаем игру
    ...
    }
```

И функция загрузки:

```
bool LoadGameObject (GameObject* object)
    {
    object->image = txLoadImage (object->imageFile);

    if (!object->image)
        {
        txMessageBox ("Не загрузилась картинка");
        return false;  // Функция завершилась неуспешно, картинка не загрузилась, играть нельзя
        }

    object->sizeX = txGetExtentX (object->image);
    object->sizeY = txGetExtentY (object->image);

    txUseAlpha (object->image);

    return true;  // Функция завершилась успешно, будем играть (если разрешит учитель
информатики).
    }             // Погодите... это же не игра, это отладка... куда же без нее...
```

Плюсом этого подхода является то, что функция может по-прежнему возвращать значение, но уже не структурного типа, а логического, и означать это будет успех или неуспех загрузки картинки. Дополнительная диагностика никогда не помешает.

Последнее, что можно доделать, это более ясная информация, выдаваемая при ошибке загрузки. Сейчас, если изображение не загрузилось, функция txMessageBox выдает сообщение "Не загрузилась картинка" и ничего не говорит об имени файла, который не загрузился. Если картинок много, будет непонятно, в чем проблема. Как включить в сообщение значение переменной object->imageFile, содержащей имя файла?

Проще всего это сделать через функцию printf, способную вставлять значения переменных в сообщение:

```
    printf ("Не загрузилась картинка из файла '%s'", object->imageFile);
```

Но printf выводит надпись в не очень красивом виде. С другой стороны, txMessageBox не позволяет комбинировать текст и переменные, а берет уже готовую строку, где значения переменных уже вставлены.

Один из эффективных способов сделать это такой:

- Сделаем временную переменную, способную хранить текстовую строку определенной длины, например, 256 символов. (Она способна поместить в себя 255 любых полезных символов, и еще один специальный нулевой символ нужен, чтобы обозначать конец строки. Компилятор и стандартные функции Си ставят этот символ автоматически);
- Воспользуемся функцией, родственной принтфу – sprintf, она не печатает надпись на экран, а помещает ее в строку (которую мы ранее создали), и с этой строкой потом можно делать что угодно;
- Передадим эту строку в функцию txMessageBox.

Как это записать?

```
    if (!object->image)
        {
        char tempString [256] = "";
        sprintf (tempString, "Не загрузилась картинка из файла '%s'", object->imageFile);
        txMessageBox (tempString);

        return false;
        }
```

Что будет, если суммарная длина поясняющего текста и имени файла будет больше длины строки tempString, то есть sprintf "напечатает" в строку tempString больше 255 символов? К сожалению, sprintf никак не проверяет емкость строки, и поведение программы будет неопределенным. Лучше воспользоваться более безопасной функцией snprintf, но она есть не во всех версиях стандартной библиотеки. Попробуйте сделать один из таких вызовов:

```
    sprintf (tempString, 256, "Не загрузилась картинка из файла '%s'", object->imageFile);

    sprintf (tempString, sizeof (tempString),
             "Не загрузилась картинка из файла '%s'", object->imageFile); // Этот вызов лучше. Почему?
```

Если это скомпилировалось, то лучше использовать snprintf.

Таким же образом, кстати, можно работать с функциями txTextOut и txDrawText, которые тоже не умеют вставлять значения переменных, как и txMessageBox. С помощью sprintf/snprintf и txTextOut/txDrawText можно сделать, например, табло счета игры, где будет отображаться оставшееся время или набранные очки.

Подытоживая, проектирование функций лучше вести так:

1. Вовремя заметить, что какой-то кусок кода имеет собственный смысл и может быть полезен для будущего;
2. Выделить его в функцию;
3. "Поиграть" с ее возможными вызовами, добавляя и убирая возможные значения параметров (при этом функция еще не написана);

4. Если способ вызова устраивает, то написать функцию, приняв подходящие параметры в нужной форме (по указателю или нет).

Когда пишете функцию:

- Если функция создает внутри себя "главный результат" – вычисленное значение, структурную переменную, еще что-то – то вернуть ее значение вызывающей стороне.
- В качестве результата можно использовать признак успешного или неуспешного завершения функции, который потом можно проверить на вызывающей стороне и вовремя среагировать на ошибки.
- В случае ошибки функция должна выдавать подробную информацию, что в ней не так, чтобы быстро найти причину проблемы.

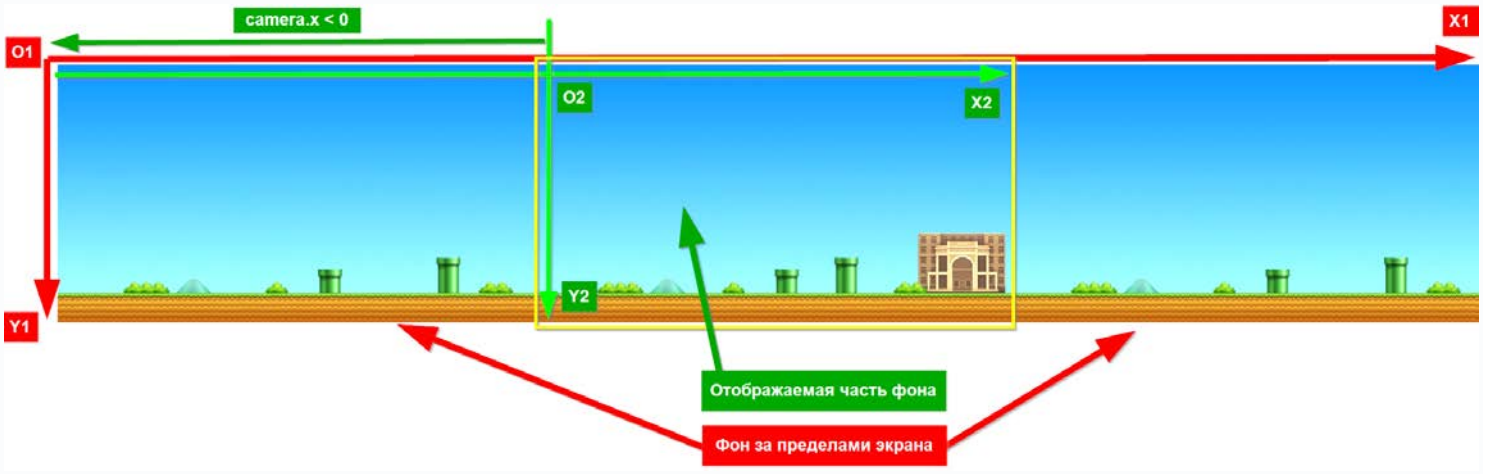
2.8. Относительная система координат

Во многих играх можно перемещаться по игровому полю в разные стороны за пределы экрана. Если персонаж бежит слева направо, добегает до правого края окна и продолжает движение, то все объекты игры, включая фон, перемещаются налево, давая возможность бежать дальше. То есть игровой мир получается больше, чем его часть, отображающаяся через окно.

Проще всего сделать это, нарисовав огромную картинку фона, существенно большую, чем ширина окна, и отображать ее на экран, "пропуская" ее некоторую часть слева. Это легко сделать, указав в вызове функции отображения фона x-координату верхнего левого угла копируемого изображения не равной нулю. Эта координата будет меняться, если Марио будет натыкаться на правую или левую границу окна.

То есть мы получаем относительную (большую) систему координат, связанную с фоном, которая отображается в абсолютную (меньшую) систему координат, связанную с окном программы. Большая система будет "прокручиваться" относительно меньшей при некоторых движениях Марио.

Все координаты объектов игры мы будем отсчитывать от большей системы координат, а "вспоминать" про меньшую – только при отображении объектов на экран (рисовании). Соответственно, это затронет все рисовательные функции.



На рисунке выше $O_1X_1Y_1$ – это глобальная система координат, связанная с фоном, а $O_2X_2Y_2$ – локальная, связанная с экраном. Так как TXLib работает только с экранной системой координат $O_2X_2Y_2$ и ничего не знает о том, что мы ввели глобальную систему $O_1X_1Y_1$, то

координаты левого верхнего угла фона при вызове функция копирования мы будем передавать отрицательные, чтобы эти функции сместили фон влево.

Можно также не рисовать длинный фон, а сделать так, чтобы более короткая картинка бесконечно повторялась, этот вариант мы рассмотрим ниже.

Как все это выразить? В разработке игр за отображение игрового мира обычно отвечает объект “камера”. Она определяется, как минимум, той самой x-координатой левой части фона, которая определяет, какую часть фона показывать, и которая меняется, если главный персонаж “выбегает” из экрана. Для общности, на будущее, заведем еще и y-координату. С ее помощью можно будет сделать смещение фона по высоте, например, при прыжках.

Итак:

```
struct Camera
    {
    double x, y;
    };
```

Функции, работающие с камерой, будут такие:

1. Смещение камеры при “выбегании” Марио за пределы экрана;

2. Все функции отображения на экран:
    a. Рисование фона;
    b. Рисование игровых объектов.

Смещение реализуем функцией (конечно же), работающей со структурой (конечно же). “Выбегание” у нас бывает двух типов:

1. Налево. Марио оказывается левее левой границы окна (в большей системе координат, связанной с фоном). Смещаем координату камеры налево, чтобы Марио смог отобразиться в окне. Фон при этом сместится вправо.

2. Направо. Марио оказывается правее правой границы окна. Марио оказывается правее правой границы окна (в большей системе координат, связанной с фоном). Смещаем координату камеры направо, чтобы Марио смог отобразиться в окне. Фон при этом сместится влево.

```
void MoveCamera (Camera* camera, GameObject mario, int sizeX)
    {
    if (mario.x < camera->x)         camera->x = mario.x;          // Левее левой границы

    if (mario.x > camera->x + sizeX) camera->x = mario.x - sizeX;  // Правее правой границы
    }
```

Как отображается фон? В простейшем случае так:

```
void DrawBack (HDC megaBackImage, int scrSizeX, Camera camera)
    {
    txSetFillColor (TX_BLACK);  // На всякий случай, если фон отобразится неправильно.
    txClear();                  // При правильной работе фона это не нужно.

    txBitBlt (-camera.x, 0, megaBackImage);
    }
```

Параметр megaBackImage отвечает за фон, ширина которого много больше, чем ширина окна. Если Марио забежит за правую границу окна, фон вместе с Марио сместится налево и большой ширины фона хватит, чтобы отобразиться в окне.

Но, кстати, если Марио забежит левее левой границы окна, то фон начнет смещаться направо, а слева у фона у нас нет запаса ширины, и мы увидим там черное пространство с уезжающим направо фоном. Это можно “подкостылить”, заранее сместив фон налево, чтобы создать “запас смещения”, но лучше подумать о том, как бесконечно “размножать” фон при смещении в любую сторону.

Рисование героев тоже станет зависеть от положения камеры:

```
void DrawGameObject (GameObject object, Camera camera)
    {
    ...
    txCircle (object.x - camera.x, object.y - camera.y, 5);

    txAlphaBlend (object.x - object.sizeX/2 - camera.x,
                  object.y - object.sizeY/2 - camera.y, object.image);
    }
```

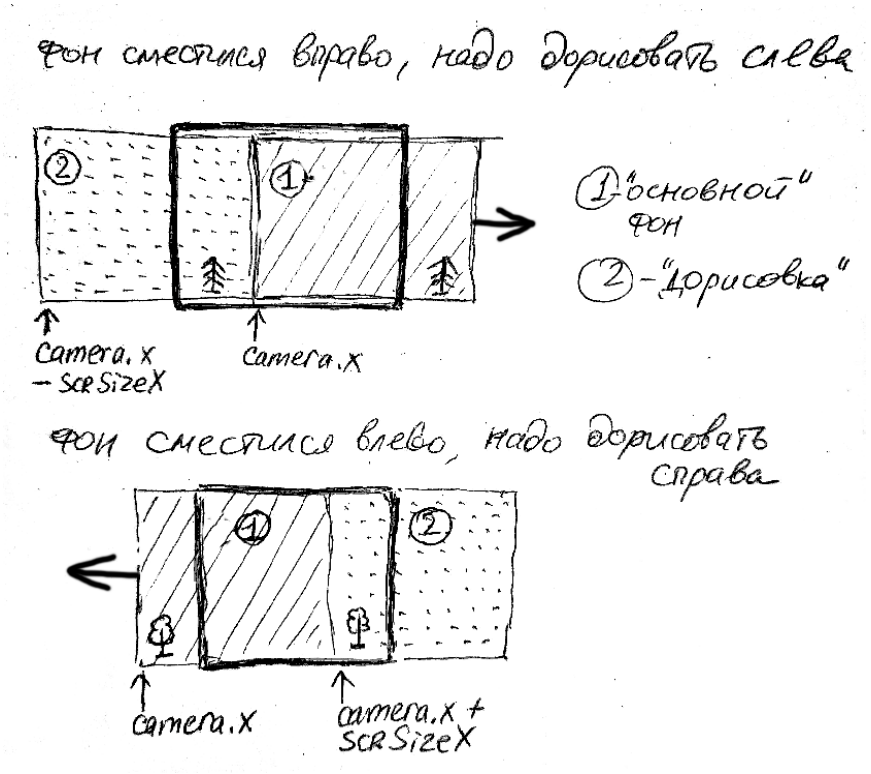
Попробуйте это.

Идею камеры можно использовать и для других эффектов движения персонажа: например, при разгоне камера плавно смещается чуть назад, при торможении – чуть вперед, при прыжке – чуть вниз, а затем плавно возвращается обратно. Это иногда называется инерционной камерой. Добиться этого можно, рассматривая камеру как игровой объект – не только с координатами, а и со скоростями и с ускорениями. Можно даже и с массой. При управлении персонажем на него фактически воздействуют силы, и аналогичные силы можно прикладывать и к камере. Чтобы камера не “болталась”, ее можно “закрепить” “резинкой” к некоторой точке через закон Гука, она будет постоянно воздействовать на камеру, притягивая ее к закрепленной точке. Попробуйте, смотрится весьма прикольно.

Теперь давайте подумаем, как обойтись без мега-фона (как бы это не было обидно некоторым операторам сотовой связи), чтобы обычная картинка фона “размножалась” сама при смещении камеры. Ну то есть, конечно, не сама, это мы ее будем размножать.

Во-первых, нужно взять фон, размером с окно, такой, чтобы у него левый край бесшовно стыковался с правым, то есть картинка в левой части фона должна повторяться в правой. Если бы вы свернули такой фон в цилиндр, на стыке вы бы не увидели границы. Проще всего в графическом редакторе скопировать кусок из левой части фона направо. Или взять левую половину фона и отразить ее по горизонтали направо и поместить на место правой половины, интересный эффект получается.

Во-вторых, поймем, сколько таких фонов потребуется. То есть сколько раз достаточно скопировать фон в окно, чтобы полностью покрыть площадь окна при любых смещениях. Сначала разберем небольшие смещения:

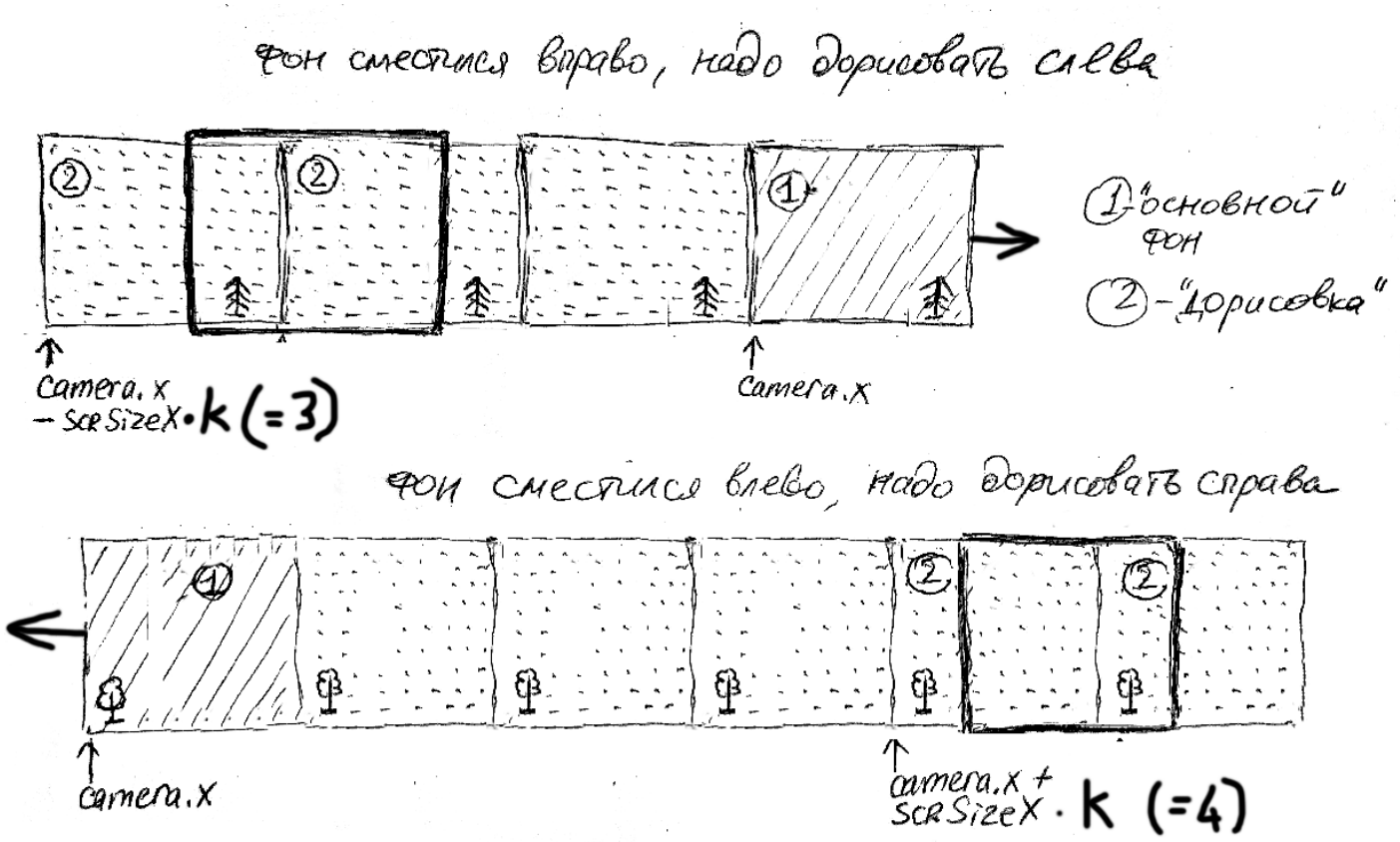


То есть достаточно двух копирований фона на экран, отличающихся на ширину экрана. Можно написать так:

```
void DrawBack (HDC megaBackImage, int scrSizeX, Camera camera)
    {
    txSetFillColor (TX_BLACK);  // На всякий случай, если фон отобразится неправильно.
    txClear();                  // При правильной работе фона это не нужно.

    txBitBlt (-camera.x, 0, megaBackImage);

    txBitBlt (-camera.x - scrSizeX, 0, megaBackImage);  // Если смещение вправо
    txBitBlt (-camera.x + scrSizeX, 0, megaBackImage);  // Если смещение влево
    }
```

Тут три копирования, и каждый раз одно лишнее. Подумайте, как обойтись двумя копированиями.

Более сложный случай, если Марио сместился на большое расстояние, значительно большее, чем ширина экрана. Тогда мы можем сделать так. Если смещение получилось слишком вправо, "приведем" его к меньшему интервалу: картинка ведь повторяется, поэтому вычтем из координаты целое число ширин экрана, скажем, k * scrSizeX:



Тут уже на экране не будет "основного фона", он слишком далеко уехал в сторону. Тут будут две "дорисовки", одна правее, одна левее. Чтобы найти их координаты, не определяя, чему равно k, ниже это написано в виде цикла. Вы можете оптимизировать код, избавившись от циклов с помощью более математичного решения.

```
void DrawBack (HDC backImage, int scrSizeX, Camera camera)
    {
    txSetFillColor (TX_BLACK);
    txClear();

    double x = -camera.x;
    while (x > 0) x -= scrSizeX;  // Если фон сместился влево,  вычитаем   scrSizeX * k
    while (x < 0) x += scrSizeX;  // Если фон сместился вправо, прибавляем scrSizeX * k

    txBitBlt (x,            0, backImage);
    txBitBlt (x + scrSizeX, 0, backImage);
    }
```

Это более общий случай варианта функции, приведенной ранее. Разберитесь в нем. Иногда приходится только лишь по коду понимать, как он работает. Не всегда пишут друзья-коллеги полные комментарии, так их растак. Но вы так не делайте, пишите все избыточно ясно, раз потратили время и разобрались. Через месяц забудете, полезете в код, а там ясные комментарии, написанные вами, когда вы еще все понимали. Коллеги через плечо заглядывают и уважительно языком цокают. Красота.

Если надоел Марио, можно писать другие игры, например, FlappyBird. Она еще легче пишется. Только давайте нормально: структуры, функции, код аккуратный, названия понятные, все дела. Не надо все в мейне писать и в шифровку превращать. Упомянутые выше коллеги не поймут.

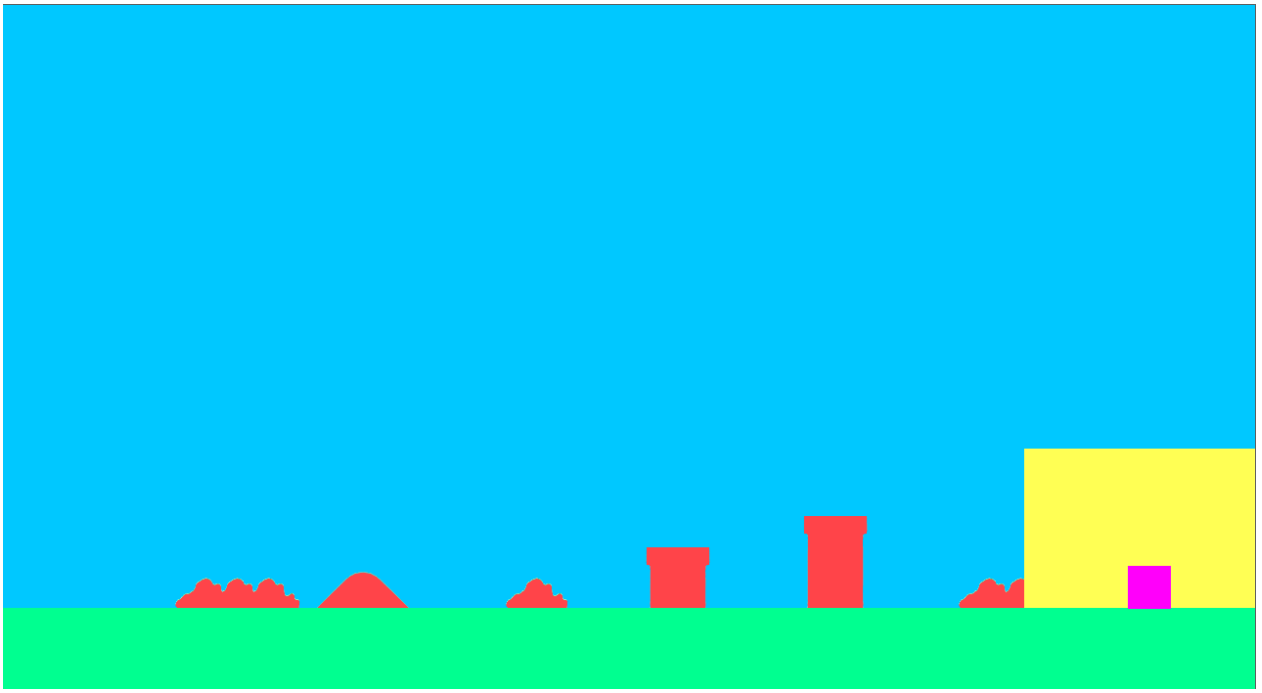
2.9. Техника "2.5D"

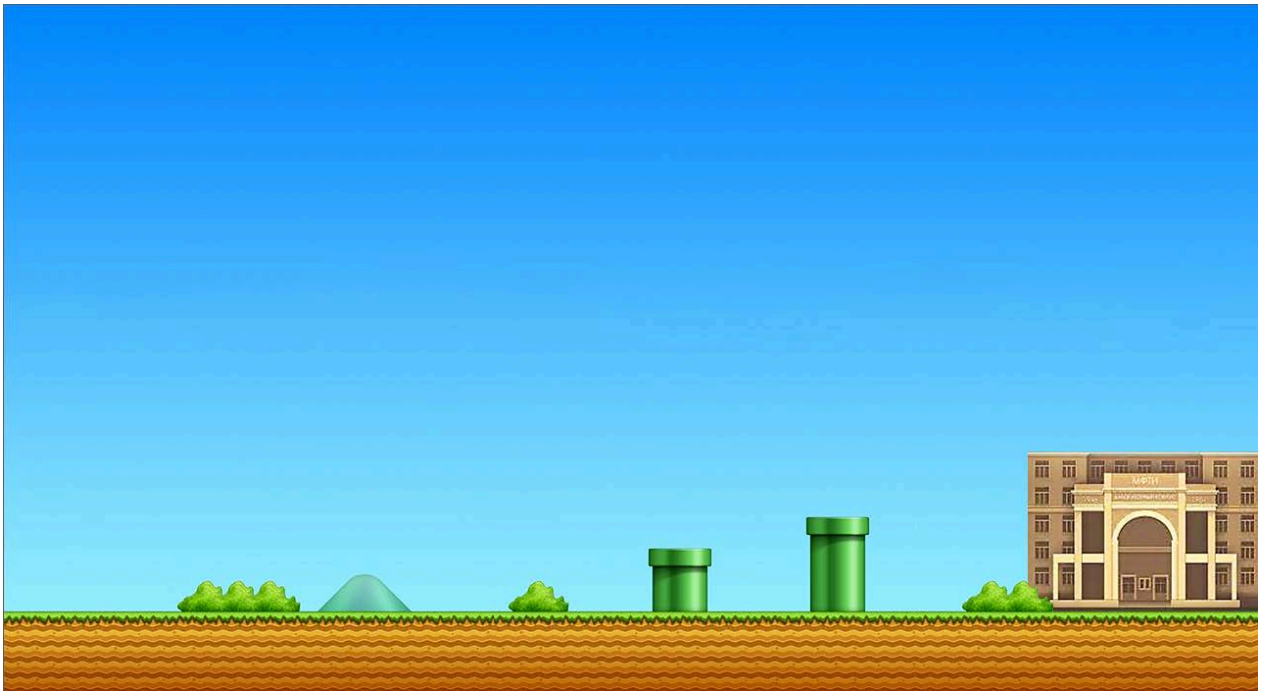
Техника "2.5D" – это набор приемов, позволяющих имитировать трехмерные сцены, используя двухмерную графику. Один из простых приемов такой. Давайте разделим фон на две части: задний план (это собственно и будет фон) и передний план. Объекты переднего плана удалим с фона и перенесем на другую картинку, оставив пространство между ними прозрачным. То есть у нас будут две картинки вместо одной и два файла. Если мы выведем на экран сначала задний план (с помощью txBitBlt), а потом сразу передний план (с помощью txAlphaBlend, чтобы учесть прозрачность), то получится то же, что и было – в окне мы увидим обычное фоновое изображение с переднеплановыми объектами. Но теперь давайте выводить передний план не сразу после заднего, а после рисования персонажей, и тогда переднеплановые объекты смогут заслонять их. Далее, при движении объекта можно смещать не только задний план, но и передний, причем с чуть разной скоростью (это называется parallax scrolling). В зависимости от соотношения смещений заднего и переднего планов получаются интересные трехмерные эффекты, напоминающие перспективу, как будто вы сами движетесь вместе с персонажем. Перспектива, конечно, не настоящая, и будет чувствоваться только при движении. Но смотреться будет прикольно. Попробуйте.

2.10. Препятствия и карты игр

Пока у нас в игре нет столкновений с препятствиями. Реализовать столкновения можно, написав функцию, принимающую координаты препятствий и их размеры (это удобно положить в структуру), а также координаты и размеры персонажа (взятые из структуры). Эта функция может рассчитывать расстояние между героем и препятствием, сравнивать его с размерами и выдавать логический результат true или false в зависимости от столкновения. В этом случае объекты будут считаться условно круглыми. Такое мы уже делали с движущимися шариками. Либо можно решить задачу столкновения двух прямоугольников, там будет больше условных операторов и придется чуть подумать, но это тоже не сильно сложно. Далее вызовем эту функцию для каждого препятствия и вуаля, столкновения в игре есть. Вообще говоря, такой способ, с "геометрической" функцией столкновения, строже и лучше, и позволяет реализовать корректный процесс отталкивания. Но он годится только для простых геометрических форм, либо надо разбивать объекты на простые формы (собственно, так и делают в игровых движках). Так можно поступать как с неподвижными, так и с движущимися препятствиями.

Для неподвижных препятствий можно применить другой способ. Он состоит в том, чтобы сделать специальную картинку – карту игры, где условными цветами будут обозначены препятствия. Вот пример карты Марио, она сверху, а снизу – фоновая картинка:

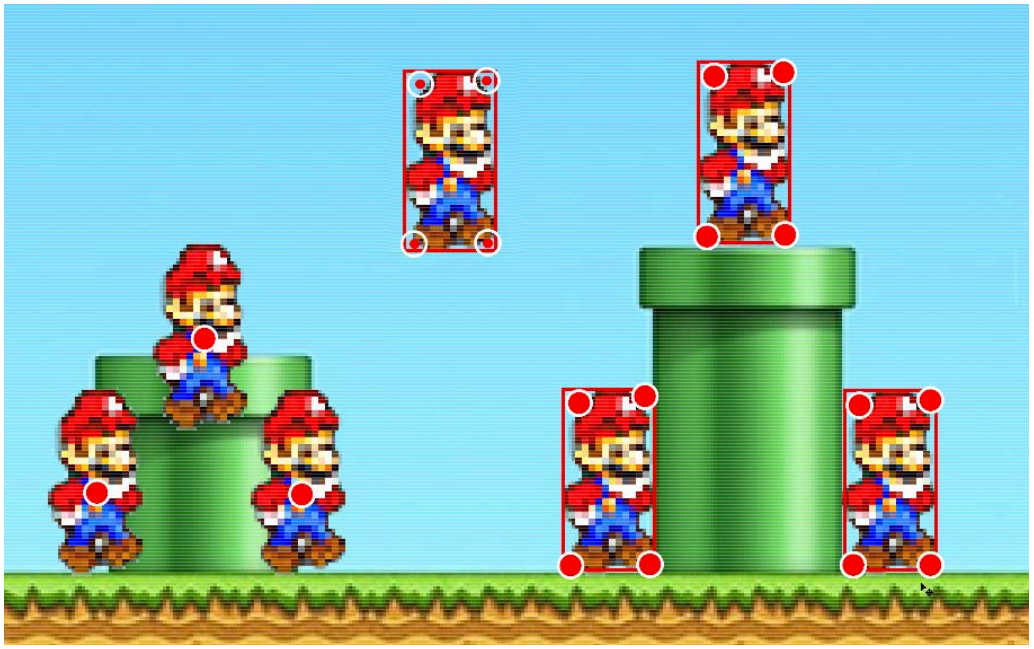




Если мы возьмем картинку с картой и выведем ее вместо фона, то увидим все препятствия в условных цветах. Кстати, так удобно делать при отладке игровых уровней. Давайте пока считать, что мы работаем с программой именно в таком режиме.

Определять, столкнулся ли Марио с препятствием, можно определять с помощью функции txGetPixel, возвращающей цвет точки экрана. Так мы уже делали с "особыми зонами" в программе с шариками. Тут, возможно, видов препятствий и условных цветов на карте будет больше, реакций на них тоже будет больше, поэтому, конечно, надо делать отдельную функцию.

Если у вас персонаж имеет сложную форму, то есть он явно не шарик, будет недостаточно одного вызова txGetPixel. Смотрите, что получится, если мы возьмем картинку Марио, и будем определять столкновения одним вызовом txGetPixel, с координатами пикселя, равными центру картинки (см. рисунок ниже, слева; для наглядности вместо карты использован фон игры):



Слева на картинке видно, что при использовании единственной точки контроля цвета Марио будет вынужден довольно сильно "наехать" на препятствие, чтобы определение столкновения сработало.

Выход – в использовании нескольких вызовов txGetPixel и комбинации результатов этих вызовов (см. рис. выше, справа). Если хотя бы один вызов дал цвет препятствия, значит, столкновение есть. Координаты txGetPixel отсчитываются от центра Марио с некоторыми смещениями в разные стороны, так, чтобы учесть форму фигуры Марио (см. рис. выше, в центре вверху). Понятно, что для такой более сложной работы со столкновениями надо сделать функцию, внутри которой будут эти несколько вызовов txGetPixel, и которая будет возвращать, было столкновение или нет.

Операция сравнения цвета, который вернула txGetPixel, и цвета препятствия на карте дает логический (булевский) результат: true (истина) или false (ложь). Если у нас четыре вызова, то получится комбинация четырех логических значений. Если их надо объединить, используется операция объединения "или" (обозначается "||"):

```
bool MarioCollisionDetection (const GameObject* mario)
    {
    bool hit = (txGetPixel (mario->x - mario->sizeX/2, mario->y - mario->sizeY/2) == TX_RED) ||
               (txGetPixel (mario->x + mario->sizeX/2, mario->y - mario->sizeY/2) == TX_RED) ||
               (txGetPixel (mario->x - mario->sizeX/2, mario->y + mario->sizeY/2) == TX_RED) ||
               (txGetPixel (mario->x + mario->sizeX/2, mario->y + mario->sizeY/2) == TX_RED);

    return hit;
    }
```

В функции выше строка с вызовом `txGetPixel (mario->x - mario->sizeX/2, mario->y - mario->sizeY/2)` содержит определение цвета в точке, соответствующей левому верхнему углу картинки с Марио, и сравнение этого цвета с цветом препятствия (красным). Остальные строчки сравнивают цвет, соответственно, правой верхней точки, левой нижней точки и правой нижней точки. Результаты комбинируются с помощью оператора "или" ("||").

В функции MarioCollisionDetection можно возвращать не логическое значение true/false, а целое число, которое покажет, с какой стороны произошло столкновение, слева или справа, или снизу, если это важно в игре.

Что делать при столкновении с препятствием? Как реагирует на это Марио? Можно по-простому отражаться от него (менять скорости на противоположные, если препятствие горизонтальное или вертикальное, и обменивать скорости по x и y, если оно наклонено под 45 градусов). Можно останавливаться (обнулять скорости). Можно при этом еще возвращаться на шаг назад, если мы вдруг оказались внутри препятствия, "заступили" за него. Возвращаться проще всего так: создайте структурную переменную oldPosition, которая будет хранить x- и y-координату персонажа до вызова функции перемещения. Если объект пересекся с препятствием, приравняйте его текущие координаты величинам, сохраненным в oldPosition. Объект тут же "вернется" в предыдущую точку, и мы не увидим, что он только что был, например, внутри стены. Конечно, до всех этих проверок, перемещений и коррекций рисовать объект не надо.

То есть у нас такой план:

1. Рисуем карту вместо фона;
2. Сохраняем старую позицию;
3. Перемещаем объект по законам физики;
4. Определяем цвет фона/карты в точке, где сейчас находится персонаж;
5. Если столкновение с препятствием – изменяем скорости, корректируем координаты, используя старую позицию;
6. Рисуем персонаж.

Прикиньте, как будет выглядеть код, соответствующий такому сценарию.

Чтобы не видеть карту, а видеть фон, можно, конечно, после определения цвета точки рисовать фон, чтобы он закрыл карту. Но это такое себе, зачем лишний раз копировать картинки на экран, это небыстрое занятие. Если внимательно прочитать документацию на функцию txGetPixel... (ах, это волшебное "внимательно прочитать"!.. Где ты в долгие бессонные ночи отладки перед релизом?.. Нет, нафиг такую романтику, мы профессионалы и ночами мы спим, а перед тем как кодить – читаем, рисуем и думаем. Пусть дилетанты ночами прогают) ...так вот, в этой документации сказано, что у этой функции есть дополнительный третий параметр dc типа HDC, раньше мы такое использовали для работы с картинками. Если указать его равным HDC загруженной картинки, то цвет будет браться из нее, а не с экрана. Возьмем пример определения столкновения с препятствием красного цвета из помощи по функции txGetPixel:

```
    if (txGetPixel (x, y) == TX_RED)         // Mess with the red -- die like the rest
        CarCrash (x, y);                     // (из какого фильма эта почти цитата?)
```

и добавим параметр dc:

```
    HDC map = txLoadImage ("MarioMap.bmp");
    ...
    if (txGetPixel (x, y, map) == TX_RED)
        CarCrash (x, y);
```

Получится определение цвета у загруженной картинки, которую мы совсем не обязаны рисовать на экране. Получается работа с невидимой картой.

Теперь можно сделать карту препятствий, взяв фон Марио, перекрасив его и сохранив в другой файл. И, наконец, убрать позорное число 590 из кода. Тем более, что оно верно только для одной картинки, и учитывает только одно препятствие – землю. Ну и пора уже убрать то, что

Марио может отталкиваться от воздуха при прыжке. Прыжок надо разрешать, только если под ногами земля.

Правда, на диске, в папке с картинками, карта будет лежать в открытом виде как файл типа BMP, его можно будет открыть любым просмотрщиком или растровым редактором и просмотреть или изменить. Можно, конечно, шифровать этот файл, но сейчас мы не будем этого рассматривать.

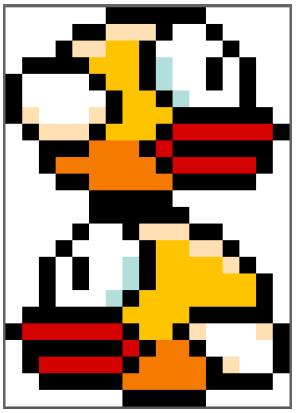
2.11. Анимация

При движении наши объекты не меняют свой вид, и это печально. Когда Марио бежит назад, то он делает это спиной вперед, рискуя упасть. Ноги его при ходьбе не двигаются. Такая себе игра, скучно.

Давайте это починим, введя анимацию персонажей, но не с Марио, а сделаем летающих туда-сюда птичек. Пусть направление полета задает то, куда повернута птичка, вправо или влево, это проще. Дальше подумаем о том, чтобы она могла махать крыльями.

Общее решение тут в том, чтобы не показывать каждый раз одно и то же изображение, а менять его от кадра к кадру. Можно, конечно, понаделать много отдельных файлов, каждый с изображением отдельной фазы движения птицы и направлением полета, но это приведет к путанице в файлах и переменных и будет неудобно. Можно ли как-то поместить все нужные изображения со всеми фазами на одну картинку?

Можно, это называется спрайтшит (spritesheet). Отдельные изображения, входящие в него, называются спрайтами. Возможно, вы видели такие спрайтшиты, когда искали картинки для Марио. Вот простой спрайтшит для птицы, учитывающий только направление ее движения:

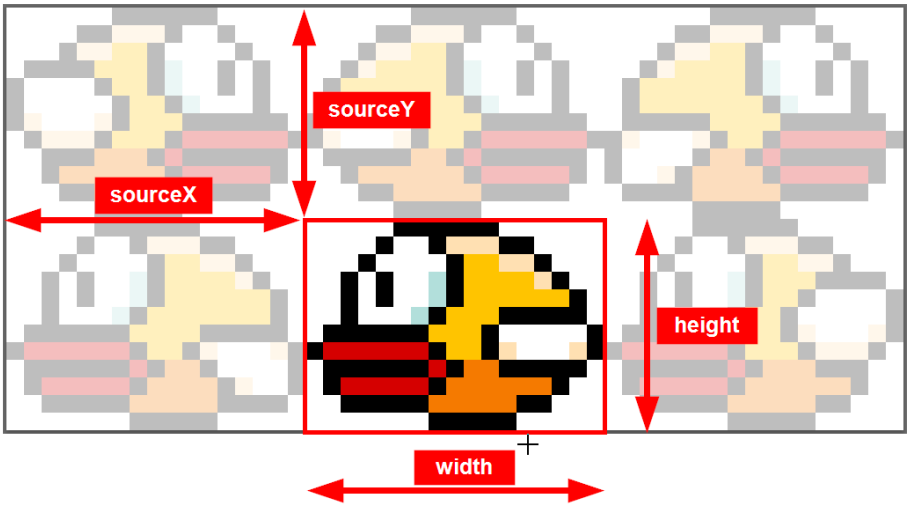


Верхняя картинка для движения слева направо, нижняя - справа налево.

Для более сложных случаев применяются картинки с большим числом кадров.

Если мы будем выводить такую картинку на экран целиком, то будет, конечно, очень странно. Но все функции копирования изображения (txBitBlt, txAlphaBlend и другие) имеют форму с большим количеством параметров, которая позволяет выводить на экран только часть картинки. Если птичка летит слева направо (ее vx > 0), то будем выводить верхнюю часть, если налево (vx < 0) – нижнюю. Посмотрите на документацию txBitBlt, она попроще. С помощью параметров xSource и ySource мы можем задать, какую часть картинки копировать. Схематически, картинка имеет собственную координатную систему, в которой мы можем задать прямоугольник, откуда будет браться копируемая часть. Координаты левого верхнего угла

этого прямоугольника и есть xSource и ySource, а его ширина и высота задаются параметрами width и height.



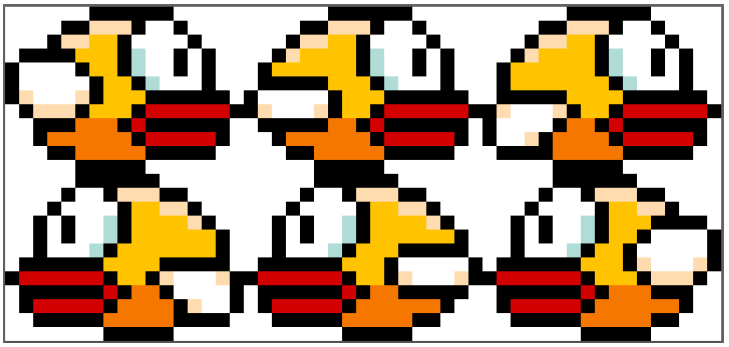
Теперь рисование птички будет вестись так (для простоты плюнем пока на то, что верхний левый угол картинки это не ее центр и передадим в txAlphaBlend координаты объекта без смещения):

```
void DrawGameObject (GameObject object)
    {
    ...

    int ySource = 0;
    if (object.vx >= 0) ySource = 0;                 // Птичка летит вправо, берем верхнюю часть
    if (object.vx <  0) ySource = object.sizeY/2;    // Птичка летит влево,  берем нижнюю  часть

    ySource = (object.vx >= 0)? 0 : object.sizeY/2;  // Другой способ выразить условие выше

    txAlphaBlend (txDC(), object.x, object.y, object.sizeX, object.sizeY/2,
                  object.image, 0, ySource);
    }
```

При столкновении с правой стенкой экрана птичка разворачивается, ее скорость по X становится отрицательной, координата ySource съезжает вниз на половину высоты картинки и выбирает нижнюю часть, отсекая верхнюю.

Правда, из-за того, что все объекты рисуются таким способом, то у Марио и облаков будут видны только их половинки. Мы это поборем, только давайте разберемся со сменой кадров анимации.

Во-первых, надо заготовить картинку с фазами движения птички. В приведенном примере ниже три фазы, расположенные по горизонтали. По вертикали – эти же три фазы, в отраженном виде, как и раньше, для обратного движения:



При каждом рисовании мы будем менять кадр по горизонтали, и тогда кадры будут чередоваться: 0, 1, 2, 0, 1, 2... (здесь нумерация кадров с нуля). Соответственно, понадобится переменная для хранения номера текущего кадра. Где она будет лежать? В структуре объекта, естественно, так как она описывает его состояние.

Еще понадобится переменная, хранящая количество кадров по горизонтали. Возможно, у разных видов объектов будет разное количество кадров, поэтому надо знать, когда счетчик кадров надо обнулять.

То есть, мы добавляем в объект переменную номера кадра frameX и количество кадров numFramesX:

```
struct GameObject
    {
    const char* imageFile;    //!< Имя файла с картинкой. Тип const char* - это текстовая строка.
    int         numFramesX;   //!< Количество кадров анимации
    ...

    HDC image;                //!< Переменная для хранения HDC изображения
    int sizeX, sizeY;         //!< Размеры изображения
    ...

    int frameX;               //!< Текущий кадр анимации
    };
```

Обратите внимание, что количество кадров задается сразу после имени файла, потому что это – свойство картинки. Текущий кадр помещен в конце структуры, чтобы не задавать его явно каждый раз при определении переменной игрового объекта, он будет автоматически обнуляться.

Функция рисования будет аккуратно рассчитывать координаты копируемой области в системе координат картинки, и вызывать функцию копирования на экран. Так как при рисовании мы меняем номер кадра, записанный в структуре, то структуру будем передавать по указателю. Не забудьте, что область копирования будет иметь меньшие размеры, чем сама картинка.

```
void DrawGameObject (GameObject* object)
    {
    ...

    int frameSizeX = object->sizeX / object->numFramesX;          // Ширина одного кадра

    int xSource     = object->frameX * frameSizeX;                // Смещение по X до текущего
                                                                  // кадра (X-координата текущего
                                                                  // кадра в системе координат
                                                                  // картинки)
    int ySource = (object->vx >= 0)? 0 : object->sizeY/2;
```

```
    txAlphaBlend (txDC(), object->x, object->y, frameSizeX, object->sizeY/2,
                  object->image, xSource, ySource);

    object->frameX++;                                                  // Увеличиваем номер кадра
    if (object->frameX > object->numFramesX) object->frameX = 0;       // Сбрасываем в 0 для
                                                                       // зацикливания

    object->frameX = (object->frameX + 1) % object->numFramesX;        // Другой способ увеличить
    }                                                                  // и зациклить
```

При каждом вызове этой функции номер кадра будет увеличиваться, а когда станет больше, чем количество кадров – сбросится в ноль. Если смена кадров будет происходить слишком быстро, сделайте отдельную функцию для смены кадра (назовите ее AdvanceAnimation, к примеру), и вызывайте в главном цикле игры менее часто. Например, на каждом десятом обороте цикла, или, лучше, по прошествии определенного времени с момента прошлого вызова. Время можно определять через функцию GetTickCount, она выдает условное время в миллисекундах:

```
void PlayMario()
    {
    ...
    int time = GetTickCount();

    while (! GetAsyncKeyState (VK_ESCAPE))
        {
        ...
        DrawGameObject (&bird);
        ...

        if (GetTickCount() - time > 100)  // Смена кадра через 100 мс
            {
            AdvanceAnimation (&bird);
            time = GetTickCount();
            }
        }

    ...
    }
```

Если вы при работе программы получаете сообщения вида "Прямоугольник копируемой области {...} не полностью лежит внутри изображения-источника {...}, функция txAlphaBlend() работать не будет", то вы неверно рассчитали координаты текущего кадра в системе координат картинки, или неверно указали ширину или высоту копируемой зоны. Это сообщение возникает обычно потому, что нижний правый угол копируемого участка, имеющий координаты xSource + width и ySource + height (в терминах системы помощи TXLib и в системе координат картинки), вылезает за пределы размеров картинки.

Из этой заготовки можно сделать неплохой анимационный движок. Можно расширить количество видов анимаций: сейчас их две – для движения вправо и влево. Если обобщить это, не привязываясь к числу 2, то можно будет добавлять другие анимации: птичка сможет кружиться на месте, танцевать и много что еще. Спрайтшиты только придется искать, но в Интернете их много. Единственно что, во всех видах анимации придется сохранять одинаковое количество кадров. Но это не такое неудобное ограничение (хотя и его можно обойти, сделав задание анимации более гибким).

2.12. Однотипные объекты. Массивы

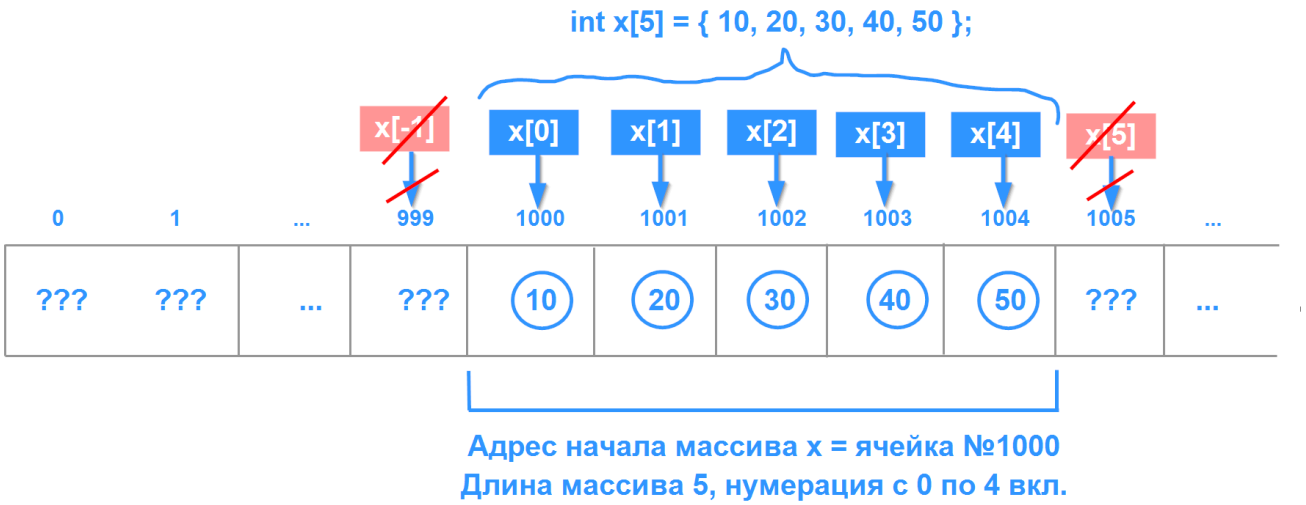
Если у нас в программе есть много однотипных объектов, например, птичек, как в примере выше, то с ними приходится совершать много одинаковых действий. Вроде бы для одинаковых действий есть циклы, но они здесь не помогают. Если у нас много однотипных переменных, они все равно объявлены отдельно друг от друга, и мы не можем устроить цикл по ним. Нам нужна какая-то группировка переменных, причем с нумерацией, чтобы можно было обратиться к ним с помощью счетчика цикла. Математики это обозначают индексами: x_0, x_1, x_2..., $bird_0$, $bird_1$, $bird_2$ и так далее. У нас, конечно, есть средство группировки переменных – структуры, но там переменные поименованы, а не пронумерованы. По именам тоже цикла не запустишь.

В программировании для этого есть такая вещь, как массивы. Массив - это одна переменная, представляющая собой последовательность пронумерованных однотипных ячеек. Каждая ячейка работает тоже как переменная, то есть у нее есть значение, которое можно изменять и т.п. Чем-то это похоже на структуру, но в структуре нет нумерации переменных и они не обязаны быть однотипными. Это потому, что структуры созданы для хранения данных, имеющих разный смысл, а массивы – для данных, похожих по смыслу. Ближайший математический аналог массива – последовательность (но там, как правило, есть формула члена последовательности, а в массиве таких закономерностей, как правило, нет), или таблично заданная функция.

Объявляется массив так:

```
int x[100] = { 10, 20, 30 };
```

Это массив из ста целых чисел – элементов массива, пронумерованных от 0 до 99. Тип всех элементов - int. Номера элементов массива называются индексами, они в языке Си начинаются с нуля (это дает более быстрый машинный код). Начальные три элемента в этом массиве с индексами 0, 1 и 2 равны соответственно 10, 20 и 30, а остальные равны нулю, так как при объявлении их значения не указаны. Вот как выглядит этот массив в оперативной памяти:



Пользоваться элементами массива просто: номер элемента указывается в квадратных скобках и может быть целым числом, переменной или арифметическим выражением.

```
x[3] = x[2] + 10;  // Третий (начиная с нуля) элемент теперь равен 40
```

Массивы просто предназначены для работы с циклами. Вот, например, как обнулить массив x:

```
int i = 0;
while (i < 100)
    {
    x[i] = 0;  // Заменяет 100 обнулений
    i++;
    }
```

А вот как задать арифметическую прогрессию:

```
int i = 0;
while (i < 100)
    {
    x[i] = 10 + 10*i;  // Заменяет 100 присваиваний
    i++;
    }
```

Или, с помощью более удобного цикла for (разузнайте про него),

```
for (int i = 0; i < 100; i++)
    x[i] = 10 + 10*i;
```

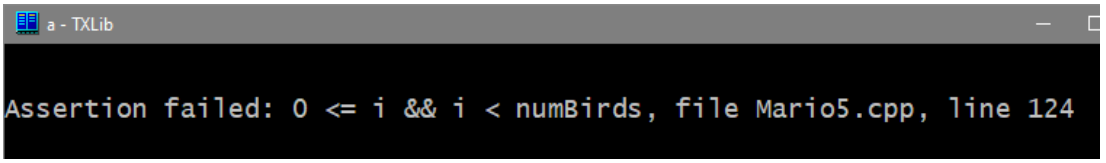
или даже так:

```
for (int i = 0; i < 100; i++) x[i] = 10 + 10*i;  // Вообще все заменяет. :)
```

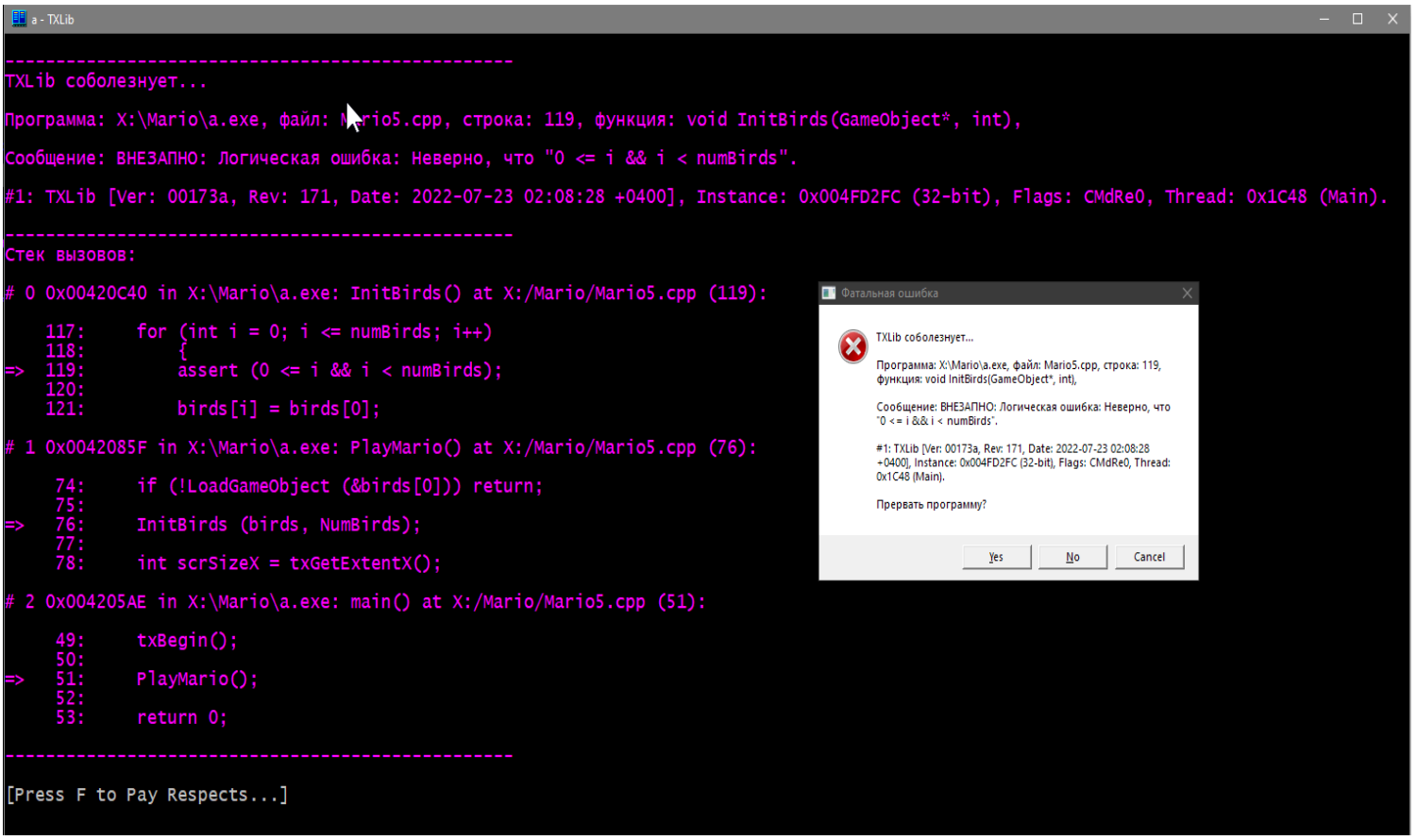
Обратите внимание, что циклы устроены так, что значение счетчика 100 в них не достигается. Потому что сотого элемента в нашем массиве не существует – помним, их нумерация идет с нуля, поэтому допустимые номера элементов – от 0 до 99 включительно. Обращение к элементам с номерами вне этого диапазона физически возможно, но приведет к неприятностям: мы получим доступ к ячейкам памяти, соседних с массивом, но не входящих в него. Там может быть что угодно: другие переменные, счетчики циклов, параметры функций, адреса возврата из них, системная информация. Изменение всего этого – неопределенное поведение программы, и еще хорошо, если мы это быстро заметим. К сожалению, компилятор и программа нас не предупредят о выходе за границы массива. Сделать это можно вручную специальной функцией assert, проверяющей критически важные условия:

```
int i = 0;
while (i <= 100)                    // Здесь специально сделана ошибка
    {
    assert (0 <= i && i < 100);     // Операция && - это логическое "И"
    x[i] = 10 + 10*i;

    assert (0 <= i-1 && i-1 < 100);
    x[i] = x[i-1] + 10;             // А здесь, кстати, тоже ошибка

    i++;
    }
```

Если условие истинно, ассерт ничего не делает. Если ложно, то сразу останавливает программу. Вот результат сработавшего ассерта в одной из программ:



Вот как это выглядит с библиотекой TXLib:



Видите, как удобно. Сразу видно, где, что и как не так.

Кто не юзает ассерт, тот ест баги на десерт!

Значение счетчика автоматически не выводится. Чтобы оно выводилось, надо печатать на экране значение счетчика, printf-ом, или специальной директивой $(i), которая выводит значение переменной i. Такая директива нестандартна, она определяется только в ТХлибе. Делать это надо до ассерта, так как он при нарушении условия не даст выполниться строчкам, лежащим ниже него.

```
int i = 0;
while (i <= 100)                // Здесь специально сделана ошибка
    {
    printf ("i = %d\n", i);     // Первый способ печати индекса
    $(i);                       // Второй способ печати индекса (только с ТХлибом)

    assert (0 <= i && i < 100); // Операция && - это логическое "И"
    x[i] = 10 + 10*i;

    i++;
    }
```

С директивой $(...) из ТХлиба можно даже так:

```
int i = 0;
while ($(i) <= 100)
    {
    ...
    }
```

Возможно, в TXLib добавится вариант ассерта специально для индексов массивов, с которым работать будет легче. Проверьте, появился ли он в свежей версии TXLib.

Все!

...не все. Если вы настоящий тру-программист, в этом месте вы должны спросить, как передавать массивы в функции. В самом деле, не будете же вы каждый раз писать кучу циклов в мейне? Фу, даже подумать так противно.

Массивы передавать в функции просто. Вот вызов:

```
int x[100] = { 10, 20, 30 };
ZeroArray (x, 100);  // Обнуляет массив
```

Вот функция:

```
void ZeroArray (int x[], int size)
    {
    int i = 0;
    while (i < size)
        {
        assert (0 <= i && i < size);
        x[i] = 0;

        i++;
        }
    }
```

Квадратные скобки после имени параметра говорят, что это массив. Размер массива, то есть количество его элементов, автоматически не передается, его надо передавать через отдельный параметр. Приняв его, функция "верит", что размер передан правильно и выполняет алгоритм и все проверки согласно ему. Неверно передать размер массива – частая и опасная ошибка.

Внутри функции мы ведем работу с параметром-массивом как обычно, как если бы он был создан внутри функции. Теперь можно не писать сто циклов в мейне, а легко вызвать сто функций. Их, правда, придется написать (и, конечно, закинуть в библиотеку), и написать документацию, но это уже другой вопрос. :)

Массивы при передаче в функции никогда не копируются – для большей скорости передачи, а вместо этого всегда передается адрес начала массива. При этом амперсанды и звездочки не пишутся, компилятор, видя передачу массива, "дописывает" их сам.

Как результат, мы можем написать функцию, которая, например, распечатывает массив, но которая внезапно из-за ошибки после распечатки его меняет:

```
void PrintArray (int x[], int size)
    {
    int i = 0;
    while (i < size)
        {
        assert (0 <= i && i < size);
        printf ("x[%d] = %d\n", i, x[i]);  // Выводит, напр.: x[5] = 50

        x[i]++;                            // Упс!
        i++;
        }
    }
```

Компилятор это разрешит, и изменится оригинал массива, потому что для массивов, даже коротких, никаких копий не создается.

Для предотвращения таких неприятностей надо явно запрещать изменение массива, если это не нужно для алгоритма:

```
void PrintArray (const int x[], int size)  // const запрещает изменение массива в функции
    {
    ...  // Остальное все так же
    }
```

Последнее, что в коде выше надо устранить – прекратить постоянно писать конкретные числа (ППППКЧ) с размерами массивов, потому что при их изменении легко пропустить какое-то конкретное число, и тогда даже ассерты не помогут, потому что будут думать, что все нормально. Если хочется ППППКЧ – а это верный признак профессионального программиста, желать ППППКЧ – то надо сверху завести константу с понятным всем названием для каждого конкретного числа. И дальше использовать только эту константу, ППППКЧ. Вот прямо сейчас во всех ваших программах ППППКЧ, замените их константами, сделайте такой рефакторинг.

Так. Массив иксов это замечательно, и неплохо подходит для математических задач, а как с массивом структур? Вот как он объявляется:

```
const int NumBirds = 10;  // Мы же ППППКЧ. Вот, константа!

GameObject birds[NumBirds] = {{"MarioBirds.bmp", 3, 100, 100,  1, 0, 0, 0, true },
                              {"MarioBirds.bmp", 3, 200, 120, -1, 0, 0, 0, true },
                              {...}};
```

Обратите внимание на двойные фигурные скобки. Внешняя пара задает массив, а каждая внутренняя пара – структуры, лежащие внутри массива. Во всех структурах, что мы не укажем явно, все поля будут равны нулю.

ОК. Посмотрим на эту конструкцию. Кое-что неудобно. Тут дело не в структурах, а в том, что у них есть одинаковые значения элементов (имена файлов, количество кадров), и копипастить это много раз – такое себе. Можно, конечно, сделать константы, и ППП... ну вы поняли. Но их все равно придется указывать для каждого элемента массива, до конца. А если объектов много?

Есть еще один момент. Не стоит одну и ту же картинку загружать много раз, это расточительно и долго. Лучше загрузить ее один раз, и как-то автоматически раскопировать ее HDC во все элементы массива. Давайте, например, нулевой объект массива считать "мастер-птичкой", для

которой будет указано имя файла и количество кадров. Мы вызовем функцию загрузки картинки только для этой мастер-птички (для нулевого элемента массива), а остальным птичкам (остальным элементам массива) приравняем полученные величины из мастер-птички (нулевого элемента).

Все это можно сделать циклом, а цикл поместить в функцию, например, InitBirds. Она будет принимать массив и количество элементов в нем. В ней мы скопируем нужные величины из нулевого элемента массива в остальные его элементы с помощью цикла.

Вот как может выглядеть вызов такой функции:

```
    GameObject birds[NumBirds] = {{"MarioBirds.bmp", 3, 100, 100, 1, 0, 0, 0, true }};

    LoadGameObject (&birds[0]);
    InitBirds (birds, NumBirds);
```

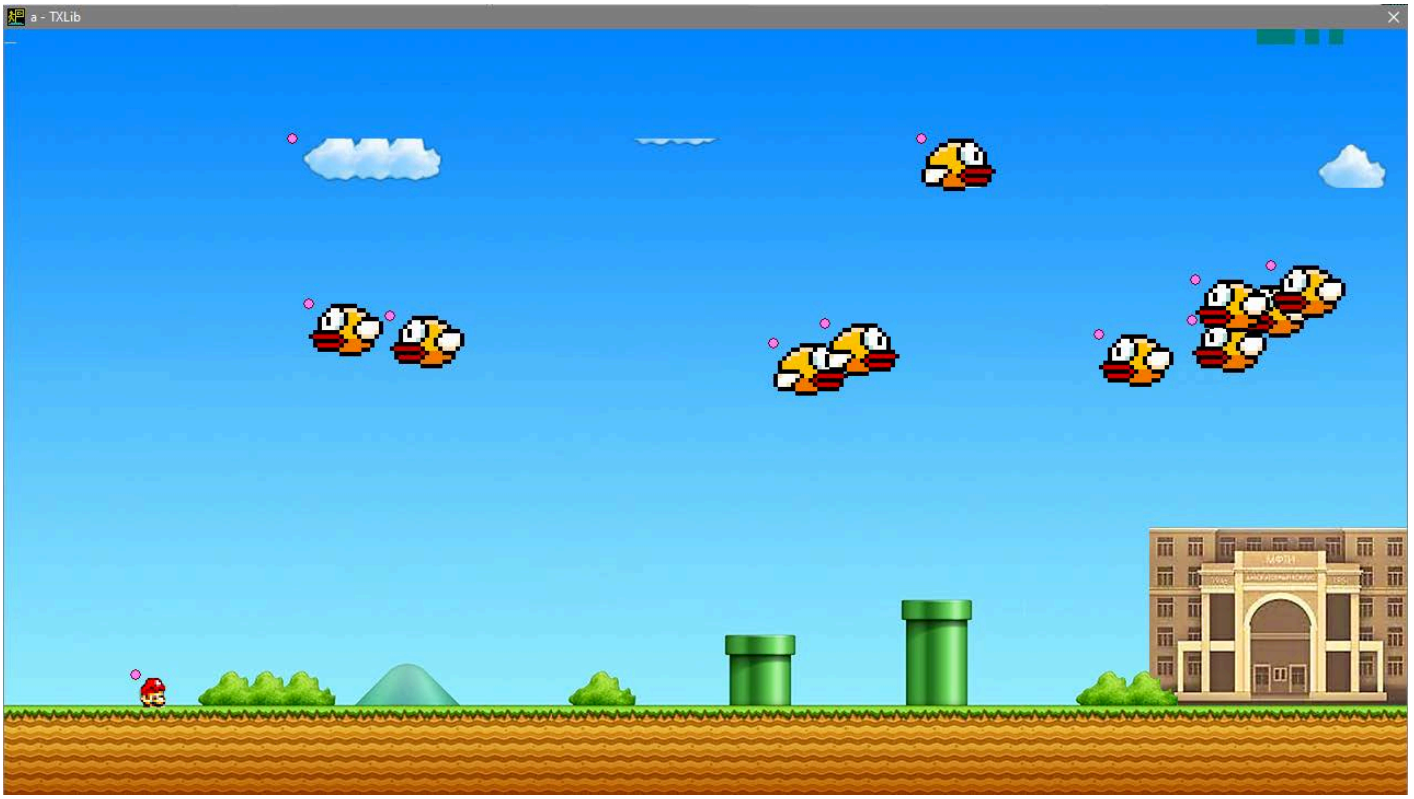
Как может выглядеть эта функция:

```
void InitBirds (GameObject birds[], int numBirds)
    {
    for (int i = 1; i < numBirds; i++)  // Нулевую птичку пропускаем
        {
        assert (0 <= i && i < numBirds);

        birds[i] = birds[0];            // Так можно приравнивать все элементы структуры почленно

        birds[i].x  = rand() % 100;     // Кое-что меняем
        birds[i].y  = rand() % 100 + 200;
        birds[i].vx = rand() % 7 - 3;
        }
    }
```

Соответственно, освобождать загруженное изображение мы будем только у мастер-птички (нулевого элемента массива), потому что у остальных элементов мы изображения не загружали, а только копировали их HDC. Освобождать много раз одно и то же единожды загруженное HDC нельзя.

Для рисования массива HDC сделаем функцию. Для перемещения – тоже сделаем функцию, в ней для птичек можно вызывать функцию физики движения облаков, так как облака ездят туда-сюда и отталкиваются от стенок. :) Далее хорошо бы подумать о столкновении с птичками, как с препятствиями. На карте их не нарисуешь, но у нас есть расстояние между объектами, оно поможет. При столкновении можно рисовать, скажем, взрыв, начислять штрафные очки и т.п.

В простейшем виде получается вот –



Облака и Марио видны наполовину, потому что эти картинки еще не переделаны под формат для анимации, а функция DrawGameObject работает со всеми объектами одинаково. И еще она у нас упрощенная – местоположение объекта задается его левым верхним углом, а не центром, поэтому все объекты съехали вниз и вправо. Но это все ерунда и легко чинится.

Смотрите, птички летают, влево-вправо сами разворачиваются, крылышками машут, код красивый работает – ляпота. Но у нас остался просто кошмарный недостаток. Игровой процесс запускается сразу, прям как не по-взрослому. Прям бах – и сразу играть. Нет, так нельзя. Пользователя надо сначала помучить выбором. Пора делать меню игры.

2.13. Меню игры

Меню – это набор экранных кнопок, в простейшем виде – прямоугольных, на которые нужно нажимать мышкой. В них нет ничего сложного, просто с клавиатурой мы немного работали, а с мышкой пока нет.

Мышь – в отличие от клавиатуры, координатный объект: у нас всегда есть координаты ее курсора. Кроме того, на мышке есть кнопки, которые можно нажимать независимо друг от друга. В дополнение к ним можно нажимать еще клавиши-модификаторы на клавиатуре: Shift, Ctr, Alt. Для простоты работы меню мы не будем различать, какие кнопки и модификаторы были нажаты, достаточно, чтобы были нажаты хоть какие-то клавиши мыши. Получить состояние тех кнопок, что на мыши, можно функцией txMouseButtons.

Получить координаты курсора мыши можно функциями txGetMouseX и txGetMouseY (а лучше, вообще-то, txGetMousePos, переходите уже целиком на структурные переменные), прочитайте про них в системе помощи TXLib. Ситуацию, когда мы кликаем внутрь кнопки, а не снаружи ее, можно отследить сравнением координат сторон кнопок и координат курсора мыши.

Кнопок несколько, все они похожи друг на друга, могут отличаться незначительно – текстом и цветом, к примеру; все действия с ними делаются более-менее одинаково (нарисовать,

проверить координаты и т.п.). Еще есть действие по выяснению номера копки, которая была нажата, если нажатие было. От этого номера зависит действие, которые мы будем делать в начале игры: запуск игры, выход из программы, показ помощи и т.п.

План наших действий таков:

1. Нарисуем все кнопки.

2. Дождемся когда пользователь нажмет на какую-то экранную кнопку мышкой, и получить ее номер. Для этого:

    a. Определим координаты курсора мыши и состояние ее кнопок;

    b. Если кнопка мыши нажата, то определим, на какой экранной кнопке находился курсор. Для этого:
        - Будем проверять принадлежность координат курсора мыши всем кнопкам по очереди.

    c. Будем повторять все это (пп. a-c), пока не нажмем на какую-то экранную кнопку;

    d. Вернем номер этой нажатой кнопки.

3. В зависимости от номера нажатой кнопки, сделаем какое-либо действие.

4. Повторяем все это (пп. 1-3), пока не захотим выйти из меню (нажмем кнопку выхода из игры).

И нам как-то это надо удобно описать и использовать. Вообще, мы уже встречались с такой ситуацией – с птичками. Да, там не было мышиного выбора, но было рисование, была физика, была расстановка птичек. Все это мы решали описанием птички как структуры и действий с птичкой как функций, работающих с этой структурой.

Так, вообще говоря, делают с любыми объектами. То есть, смотрите:

1. У нас опять есть какой-то объект (кнопка);
    a. Он описывается какими-то характеристиками;
    b. С ним выполняются какие-то действия (рисовать, проверить координаты и т.п.)

2. Мы в ответ на это снова:
    a. Делаем структуру, описывающую объект;
    b. На каждое действие пишем функцию для этого объекта;
    c. Вместо прямой работы с полями объекта стараемся все делать через эти функции.

3. Видя множество похожих объектов:
    a. Организуем из них массив;
    b. Всю работу с похожими объектами делаем через циклы, работающие с этим массивом;
    c. Для удобства эти циклы также помещаем в функции.

С птичками все прекрасно прошло. Давайте с кнопками мы теперь все сделаем по плану изначально.

Структура Button:

```
struct Button
    {
    int id;
    const char* text;
    COLORREF color;

    int x, ...
    int sizeX, ...
    };
```

id – это условный номер действия, за которое отвечает кнопка. Зачем он, если кнопки расположены в определенном порядке и у каждой есть номер? Ну а вдруг вы захотите переупорядочить кнопки, тогда номера изменятся, придется менять логику программы. Если каждая кнопка "помнит" номер своего действия, помещенного, естественно, в константу, то менять порядок кнопок в меню намного проще.

Далее идет текст кнопки, выраженный адресом, по которому находятся буквы названия кнопки. То есть название кнопки хранится "снаружи" ее. На работу с текстом это не влияет, так как мы его не меняем в ходе работы программы, что подчеркивает ключевое слово const. К сожалению, при изменении текста это уже не будет так просто, но мы пока такое рассматривать не будем.

Далее идет цвет, координаты, размеры и т.п. В принципе, кнопки можно задать и картинками, даже и с анимацией нажатия и т.п. Для простоты тоже это здесь рассматривать не будем, но вы можете запланировать это для следующей версии программы.

Ок. Всего перечисленного достаточно, чтобы задать кнопку. С данными покончено, теперь действия, то есть функции.

Во-первых, кнопку надо нарисовать:

```
void DrawButton (const Button* button)  // const - чтобы случайно не изменить кнопку при рисовании
    {
    txSetColor (TX_YELLOW);
    txSetFillColor (button->color);

    Win32::RoundRect (txDC(),
                      button->x, button->y,
                      button->x + button->sizeX, button->y + button->sizeY,
                      15, 15);  // Погуглите "Win32 RoundRect function", за что отвечают 15, 15

    txDrawText (button->x, button->y, ..., button->text);
    }
```

Функция Win32::RoundRect, не входящая в TXLib, используется тут немного для пижонства, рисования прямоугольника с закругленными углами. Можно было и обычным txRectangle обойтись. Первый параметр RoundRect, равный txDC(), это HDC изображения, где надо нарисовать прямоугольник. txDC() возвращает HDC изображения, связанного с окном программы.

txDrawText рисует текст, вписывая его в прямоугольник и центруя. Удобная функция, но чуть медленнее txTextOut.

Так как кнопки расположены в массиве, напишем функцию рисования для массива кнопок:

```
void DrawButtons (const Button buttons[])
    {
    int i = 0;
    while (buttons[i].text != NULL)  // Пока не достигнут последний, незаполненный
        {                            // элемент массива (с неуказанным текстом)
        DrawButton (&buttons[i]);
        i++;
        }
    }
```

Вы, наверное, заметили отсутствие у функции параметра размера массива и странное условие во внутреннем цикле while. Раньше в таких циклах мы писали сравнение счетчика с количеством элементов в массиве. Можно, конечно, здесь сделать так же, но тут продемонстрирован другой способ. Договоримся хранить в конце массива, в последней ячейке, элемент, который не будет использоваться для работы, но на котором будут останавливаться все алгоритмы работы с массивом. То есть, если массив представить как автомобильную дорогу, это будет знак "кирпич", после которого ехать дальше нельзя. Это позволяет не передавать в функции длину массива, а только сам массив.

При этом массив надо объявлять с дополнительным "нулевым" элементом:

```
    Button menu[] = { { 1, "Играть",     TX_GREEN, 500, 100, 300, 100 },
                      { 2, "Не играть",  TX_RED  },
                      { 3, "Не знаю :(", TX_BLUE },
                      { } };  // Элемент обозначен, но все поля его равны нулю.
                              // Это и есть нулевой элемент
```

Дополнительное удобство в том, что если при задании массива не указать его размер, но указать значения элементов в нем, то компилятор сам подсчитает элементы справа от знака "равно" и создаст массив нужного размера. И раз нулевой элемент указан в конце как {}, то с учетом его.

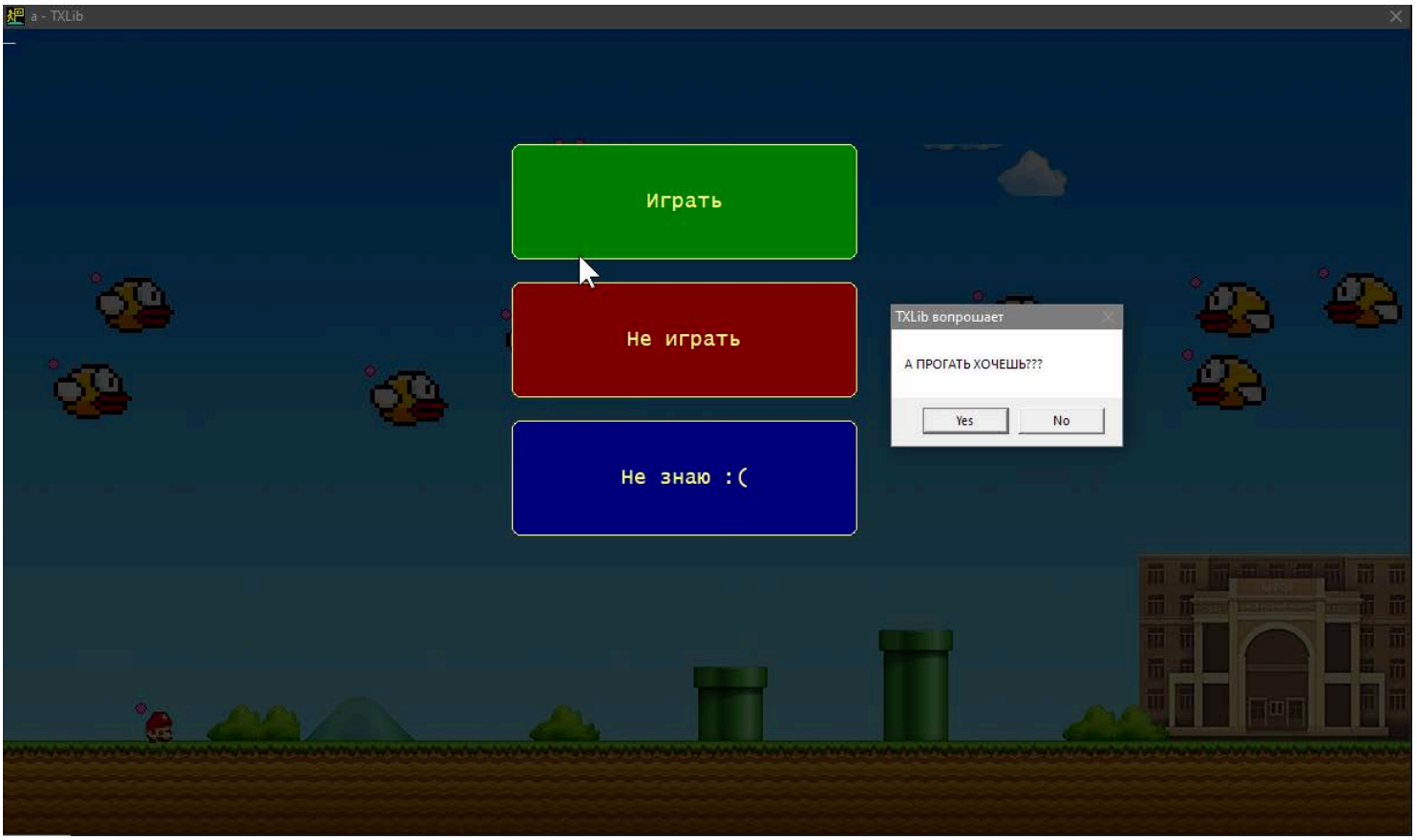
Эту же функцию можно написать с более экономным циклом for вместо цикла while (если вы его еще не знаете, то прочитайте о нем самостоятельно):

```
void DrawButtons (const Button buttons[])
    {
    for (int i = 0; buttons[i].text != NULL; i++) // Кроме того, так меньше вероятности забыть i++
        DrawButton (&buttons[i]);
    }
```

Или даже так (условие цикла можно прочитать как "пока у нас в кнопке есть текст"):

```
void DrawButtons (const Button buttons[])
    {
    for (int i = 0; buttons[i].text; i++)         // Величины, не равные и не сводящиеся к нулю,
        DrawButton (&buttons[i]);                 // считаются истинными, а нулевые - ложные
    }
```

Вот как это выглядит:



Теперь надо проверить, что координаты некоторой точки находятся внутри кнопки. Согласно нашему плану (на каждое действие мы делаем функцию), мы напишем для этого проверку в виде двух двойных неравенств:

```
bool PointInButton (const Button* button, double x, double y)
    {
    return button->x <= x && x <= (button->x + button->sizeX) &&  // Скобки для понятности
           ...  ...y... ...;
    }
```

Теперь с помощью нее можно написать функцию ожидания выбора кнопки:

```
int SelectButton (const Button buttons[])
    {
    while (! txGetAsyncKeyState (VK_ESCAPE))        // Аварийный выход из меню по нажатию Esc
        {
        double x = txMouseX(), y = txMouseY();      // Получаем координаты курсора мыши
        int pressed = txMouseButtons();             // Получаем нажатие клавиш мыши

        Sleep (10);                                 // Пауза чтобы не нагружать процессор

        if (pressed == 0)                           // Если клавиши мыши не нажаты, то...
            continue;                               // ...пропустить все ниже до конца цикла
                                                    // (цикл сразу уходит на следующий оборот)
        int i = 0;
        while (buttons[i].text != NULL)
            {
            if (PointInButton (&buttons[i], x, y))  // Если точка в кнопке, то
                return buttons[i].id;               // Вернуть ее номер и выйти из функции
            i++;
            }
        }
```

```
    return -1;                                      // Была нажата клавиша Esc
    }
```

Последнее, нужно аккуратно расставить кнопки, и не делать же это вручную. Напишем функцию, как и для птичек. Координаты начальной кнопки (с индексом 0) будут задавать левый верхний угол всего меню (в функции – x0, y0), а остальные кнопки расположатся ниже. Размеры всех кнопок будут одинаковы и также зададутся размерами начальной кнопки.

```
void InitButtons (Button buttons[])
    {
    int x0    = buttons[0].x, ...;
    int sizeX = buttons[0].sizeX, ...;

    for (int i = 1; buttons[i].text != NULL; i++)   // Начальную кнопку (i = 0) пропускаем,
        {                                           // на нулевом адресе текста останавливаем цикл
        buttons[i].x     = x0;
        buttons[i].y     = y0 + (sizeY + sizeY/5) * i; // y0 + (размер + расстояние между кнопками)
        buttons[i].sizeX = sizeX;
        buttons[i].sizeY = sizeY;
        }
    }
```

Определив структуру Button и написав удобный функции для нее и для массива кнопок, можно всем этим удобно пользоваться:

```
    ...

    Button menu[] = {{ 1, "Играть",      TX_GREEN, 500, 100, 300, 100 },  //! @TODO id = 1,2,3
                     { 2, "Не играть",   TX_RED   },                      //! срочно в константы!
                     { 3, "Не знаю :(", TX_BLUE  },                       //! Пусть END - такая
                     {END}};                                              //! константа, равная 0

    InitButtons (menu);

    HDC black = txCreateCompatibleDC (1280, 700);

    while (! txGetAsyncKeyState (VK_ESCAPE))     // На всякий случай, выход по Esc
        {
        txAlphaBlend (0, 0, black, 0, 0, 0.75);  // Полупрозрачное стирание (fading)

        DrawButtons (menu);

        int id = SelectButton (menu);            // id = номер команды или -1 в случае Esc

        if (id == 1)                             //! @TODO Срочно в константу!
            {
            PlayMario();

            while (txGetAsyncKeyState (VK_ESCAPE)) Sleep();  // Если вы вышли из игры по нажатию
            }                                                // Esc, то ждем отпускания клавиши

        if (id == 2)                             //! @TODO Срочно в константу!
            {
            if (txMessageBox ("А ПРОГАТЬ ХОЧЕШЬ???", "TXLib вопрошает", MB_YESNO) == IDYES) continue;
            txMessageBox ("А ЗРЯ...", "TXLib не одобряет", MB_ICONHAND);
            }

        if (id == 3)                              //! @TODO Срочно в константу!
            {
            txMessageBox ("НАДО ХОТЕТЬ:\n\n"
                          "1) ПРОГАТЬ\n"
```

```
                        "2) БОТАТЬ ФИЗИКУ\n"
                        "3) БОТАТЬ МАТЕШУ\n"
                        "4) БОТАТЬ РУССКИЙ (помни о ЕГЭ!)\n"
                        "5) ПОСТУПАТЬ НА ФИЗТЕХ!!!", "ГАЙД ЧТО ХОТЕТЬ", MB_ICONHAND);
            break;
            }
        }

    ...
```

Код выше надо зарефакторить: во-первых и главных, убрать "магические числа" 1, 2, 3, -1 и так далее в константы с понятными именами, например, CMD_PLAY, CMD_NOPLAY и т.п., это прямо самое главное. Лучше для этого воспользоваться оператором enum, предназначенным для создания серии родственных констант, прочитайте про его синтаксис самостоятельно. Дальше, вместо серии ифов лучше использовать оператор switch, он немного компактнее, однако придется решить небольшой вопрос с оператором break внизу кода выше.

2.14. Пример игры

Вот полный пример известной игры про волка, ловящего яйца. В нем не все идеально, постарайтесь найти в нем, что можно улучшить. Пожалуйста, используйте его для изучения, а не для игры на уроках. :)

```
//=========================================================================================
//! @file WolfGame.cpp
//!
//! @mainpage
//! Игра про волка, который ловит яйца.
//! Пример игры на библиотеке TXLib в учебном пособии "Программирование как проектирование".
//!
//! @author Ilya Dedinsky
//! @date   2023
//!
//! @par Управление:
//!    - Стрелка влево    - Волк ловит левое верхнее яйцо
//!    - Стрелка вправо   - Волк ловит верхнее правое яйцо
//!    - Клавиша End      - Волк ловит левое нижнее яйцо
//!    - Клавиша PgDn     - Волк ловит правое нижнее яйцо
//!    - Стрелка вверх    - Волк переключается на ловлю верхних яиц
//!    - Стрелка вниз     - Волк переключается на ловлю нижних яиц
//!    - Esc              - Выход из игры
//!
//! Файлы:
//!    - WolfGame.cpp     - Исходный текст игры
//!    - TXGameEngine.cpp - Игоровой недодвижок
//!
//! @image html WolfGame.png
//=========================================================================================

#include "TXLib.h"

//-----------------------------------------------------------------------------------------

const bool Debug       = false;  //!< Режим отладки. Если включен, рисуются ограничивающие
                                 //!< прямоугольники (bounding boxes) объектов
```

```
//-----------------------------------------------------------------------------------------

#include "TXGameEngine.cpp"      //   Да, такое пафосное название

//-----------------------------------------------------------------------------------------

const int  MaxTime     = 60;     //!< Максимальное время игры в секундах
const int  MaxFailures = 100;    //!< Максимальное количество разбитых яиц

const int  NumChickens = 4;      //!< Количество куриц
const int  NumEggs     = 4;      //!< Количество одновременно катящихся яиц

enum WolfAnimations              //!  Анимации волка
    {
    WOLF_LEFT_UP    = 0,         //!< Волк повернут влево,  корзина вверху
    WOLF_RIGHT_UP   = 1,         //!< Волк повернут вправо, корзина вверху
    WOLF_LEFT_DOWN  = 2,         //!< Волк повернут влево,  корзина внизу
    WOLF_RIGHT_DOWN = 3,         //!< Волк повернут вправо, корзина внизу
    WOLF_DANCE      = 4          //!< Волк танцует
    };

enum EggAnimations               //!  Анимации яйца
    {
    EGG_GOING       = 0,         //!< Яйцо катится
    EGG_OK          = 1,         //!< Яйцо катится и сейчас попадет в корзину к волку
    EGG_FAILED      = 2          //!< Яйцо катится и сейчас разобьется
    };

//=========================================================================================

void Play();
void DoDance     (GameObject* wolf, HDC back);
void DrawScore   (int score, int time);
void DrawBanner  (COLORREF color, const char* text);

void ControlWolf (GameObject* wolf);
int  CheckEgg    (GameObject* egg, int chicken, GameObject* wolf, double dt);
int  CheckEggs   (GameObject  eggs[], int numEggs, GameObject* wolf, double dt);
void RespawnEgg  (GameObject* egg, double dt);

//=========================================================================================
//! Главная функция программы
//=========================================================================================

int main (int argc, const char* argv[])
    {
    txCreateWindow (800, 600);
    txTextCursor (false);

    txSelectFont ("Comic Sans MS", 50);

    srand (time (NULL));

    if (!(argc >= 2 && strcmp (argv[1], "--nosound") == 0))
        {
```

```
        txPlaySound ("Banjo.wav", SND_ASYNC | SND_LOOP);

        DrawBanner (TX_BLUE, "3");        Sleep (1000);
        DrawBanner (TX_BLUE, "2");        Sleep (1000);
        DrawBanner (TX_BLUE, "1");        Sleep (1000);
        DrawBanner (TX_BLUE, "START!!!"); Sleep (1000);
        }

    Play();

    txPlaySound (NULL);
    }

//---------------------------------------------------------------------------------------------
//! Функция главного цикла игры
//---------------------------------------------------------------------------------------------

void Play()
    {
    HDC backImage = txLoadImage ("Back.bmp");
    if (!backImage) { txMessageBox ("Не могу загрузить 'Back.bmp'"); return; }

    GameObject wolf                  =  {{ 400, 370 },         "Wolf.bmp",    2, 5 };

    GameObject eggs    [NumEggs]     = {{{  50, 265,  10, 2 }, "Egg.bmp",     5, 3 },
                                          {{ 755, 265, -10, 2 }},
                                          {{  50, 385,  10, 2 }},
                                          {{ 755, 385, -10, 2 }}};

    GameObject chickens[NumChickens] = {{{  50, 190 },         "Chicken.bmp", 2, 1 },
                                          {{ 750, 190 }},
                                          {{  50, 330 }},
                                          {{ 750, 330 }}};

    if (!LoadObject (&wolf))        return;
    if (!LoadObject (&eggs[0]))     return;
    if (!LoadObject (&chickens[0])) return;

    double dt = 1;
    int time = 0;

    int failed = 0;

    ResetObject (&wolf);

    ResetObjects (eggs, NumEggs);
    for (int i = 0; i < NumEggs; i++) RespawnEgg (&eggs[i], dt);

    ResetObjects (chickens, NumChickens);
    for (int i = 0; i < NumChickens; i++) chickens[i].frame = rand() % chickens[i].numFrames;

    txBegin();

    int startTime = GetTickCount();

    while (! txGetAsyncKeyState (VK_ESCAPE))
```

```
        {
        txBitBlt (0, 0, backImage);

        int timeLeft = MaxTime - (GetTickCount() - startTime) / 1000;
        if (timeLeft < 0) timeLeft = 0;

        failed += CheckEggs (eggs, NumEggs, &wolf, dt);

        DrawScore (failed, timeLeft);

        if (timeLeft <= 0)
            break;

        if (failed >= MaxFailures)
            break;

        DrawObject  (&wolf);

        DrawObjects (eggs,     NumEggs);
        DrawObjects (chickens, NumChickens);

        ControlWolf (&wolf);

        BaseMoveObjects (eggs, NumEggs, dt);

        txSleep (100);

        time++;
        }

    txEnd();

    txSelectFont ("Comic Sans MS", 56, 25, 10);

    if (failed >= MaxFailures)
        {
        DrawBanner (TX_BLACK, "Game OVER :((");

        txPlaySound ("Quack.wav", SND_SYNC);
        }
    else
        {
        DoDance (&wolf, backImage);

        txDisableAutoPause();
        }

    txDeleteDC (backImage);
    txDeleteDC (wolf.image);
    txDeleteDC (eggs[0].image);
    txDeleteDC (chickens[0].image);
    }

//=============================================================================================
//! Функция управления волком
//!
```

```
//! @param wolf Волк
//=============================================================================================

void ControlWolf (GameObject* wolf)
    {
    if (GetAsyncKeyState (VK_LEFT))  wolf->animation = WOLF_LEFT_UP;
    if (GetAsyncKeyState (VK_RIGHT)) wolf->animation = WOLF_RIGHT_UP;
    if (GetAsyncKeyState (VK_END))   wolf->animation = WOLF_LEFT_DOWN;
    if (GetAsyncKeyState (VK_NEXT))  wolf->animation = WOLF_RIGHT_DOWN;

    if (GetAsyncKeyState (VK_UP))    wolf->animation &= ~WOLF_LEFT_DOWN;
    if (GetAsyncKeyState (VK_DOWN))  wolf->animation |=  WOLF_LEFT_DOWN;
    }

//---------------------------------------------------------------------------------------------
//! Функция танца волка
//!
//! @param wolf Волк
//! @param back Фоновая картинка игры
//---------------------------------------------------------------------------------------------

void DoDance (GameObject* wolf, HDC back)
    {
    while (kbhit()) getch();

    wolf->cur.x = 800/2;
    wolf->cur.y = 600/2;
    wolf->animation = WOLF_DANCE;

    txBegin();

    for (int time = 0; ; time++)
        {
        txBitBlt (0, 0, back);

        DrawBanner (TX_RED, "You won!");

        DrawObject (wolf);

        if (kbhit()) { getch(); break; }

        txSleep (150);
        }

    txEnd();
    }

//=============================================================================================
//! Функция проверки всех яиц на попадание в корзину
//!
//! @param eggs    Массив яиц
//! @param numEggs Количество яиц в массиве
//! @param wolf    Волк
//! @param dt      Шаг времени
//=============================================================================================
```

```
int CheckEggs (GameObject eggs[], int numEggs, GameObject* wolf, double dt)
    {
    int failed = 0;

    for (int i = 0; i < numEggs; i++)
        failed += CheckEgg (&eggs[i], i, wolf, dt);

    return failed;
    }

//-------------------------------------------------------------------------------------------
//! Функция проверки одного яйца, снесенного курицей, на попадание в корзину
//!
//! Если яйцо укатилось довольно далеко от точки появления,  то яйцо либо попало в корзину, либо
//! разбилось.
//!
//! Если и номер курицы, которая его снесла, равен номеру анимации волка, то волк сейчас обращен
//! к этой курице, и яйцо попало в корзину. Если не равен, то волк работает с другой курицей
//! и проверяемое яйцо разбилось.
//!
//! @param egg     Проверяемое яйцо
//! @param chicken Номер курицы
//! @param wolf    Волк
//! @param dt      Шаг времени
//-------------------------------------------------------------------------------------------

int CheckEgg (GameObject* egg, int chicken, GameObject* wolf, double dt)
    {
    double dist = Distance (egg->cur, egg->init);

    if (100 <= dist && dist <= 140)
        {
        if (wolf->animation == chicken)
            egg->animation = EGG_OK;
        else
            egg->animation = EGG_FAILED;
        }

    if (dist >= 140)
        {
        RespawnEgg (egg, dt);

        if (wolf->animation == chicken)
            {
            if (Debug) { $s; printf ("Chicken %d: ", chicken); $g; printf ("OK\n"); }

            return 0;
            }
        else
            {
            if (Debug) { $s; printf ("Chicken %d: ", chicken); $r; printf ("FAIL\n"); }

            return 1;
            }
        }
```

```
    return 0;
    }

//-----------------------------------------------------------------------------------------
//! Перезапуск яйца
//!
//! При перезапуске яйцо помещается в исходную точку появления (разную для разных куриц)
//! и прокатывается по наклонной плоскости на небольшое случайное расстояние.
//!
//! @param egg  Яйцо
//! @param dt   Шаг времени
//-----------------------------------------------------------------------------------------

void RespawnEgg (GameObject* egg, double dt)
    {
    ResetObject (egg);

    int t = rand() % 5;
    for (int i = 0; i < t; i++) BaseMoveObject (egg, dt);
    }

//=========================================================================================
//! Отображение табло игры
//!
//! @param failed    Количество разбившихся яиц
//! @param timeLeft  Оставшееся время в секундах
//=========================================================================================

void DrawScore (int failed, int timeLeft)
    {
    txSetColor (TX_WHITE);
    char scoreStr[100] = "";
    snprintf (scoreStr, sizeof (scoreStr), "Time Left: %ds, Eggs failed: %d", timeLeft, failed);

    DrawBanner (TX_BLUE, scoreStr);
    }

//-----------------------------------------------------------------------------------------
//! Рисование произвольной надписи на табло игры
//!
//! @param color  Цвет фона надписи
//! @param text   Отображаемый текст
//-----------------------------------------------------------------------------------------

void DrawBanner (COLORREF color, const char* text)
    {
    txSetFillColor (color);
    txSetColor (TX_NULL);  txRectangle (30, 30, 800-30, 90);
    txSetColor (TX_WHITE); txDrawText  (30, 30, 800-30, 90, text);
    }
```

```
//=========================================================================================
//! @file TXGameEngine.cpp
//!
//! Конечно, это не движок.
```

```
//! Но позволяет писать совсем простые, практически однотипные, игры.
//!
//! @author Ilya Dedinsky
//! @date   2023
//=================================================================================================

#include "TXLib.h"

//=================================================================================================

struct Point                    //!  Состояние объекта с точки зрения физики
    {
    double x,  y;               //!< Позиция объекта
    double vx, vy;              //!< Скорость объекта
    };

//-------------------------------------------------------------------------------------------------

struct GameObject               //!  Информация об игровом объекте
    {
    Point       init;           //!< Исходное состояние объекта

    const char* imageFile;      //!< Имя файла с картинкой

    int         numFrames;      //!< Количество кадров анимации (ячеек по ширине)
    int         numAnimations;  //!< Количество анимаций        (ячеек по высоте)

    Point       cur;            //!< Текущее состояние объекта

    HDC         image;          //!< Переменная для хранения HDC изображения
    int         sizeX, sizeY;   //!< Размеры изображения

    int         frame;          //!< Текущий кадр анимации (по X)
    int         animation;      //!< Текущая анимация      (по Y)
    };

//=================================================================================================

bool LoadObject      (GameObject* object);
void ResetObject     (GameObject* object);
void DrawObject      (GameObject* object);
void BaseMoveObject  (GameObject* object, double dt);

void ResetObjects    (GameObject objects[], int numObjects);
void DrawObjects     (GameObject objects[], int numObjects);
void BaseMoveObjects (GameObject objects[], int numObjects, double dt);

double Distance (Point a, Point b);

//=================================================================================================
//! Загрузка картинки объекта из файла
//!
//! @param object  Структура с информацией об объекте
//!
//! @returns Если загрузка удачна, true. Иначе false.
//=================================================================================================
```

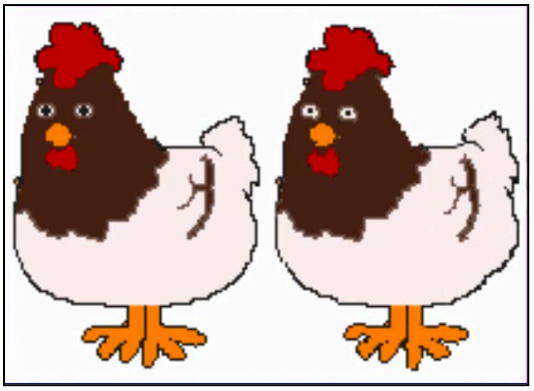
```
bool LoadObject (GameObject* object)
    {
    object->image = txLoadImage (object->imageFile);

    if (!object->image)
        {
        txMessageBox ("Не загрузилась картинка", object->imageFile);
        return false;
        }

    object->sizeX = txGetExtentX (object->image);
    object->sizeY = txGetExtentY (object->image);

    return true;
    }

//-----------------------------------------------------------------------------------------
//! Переустановка объекта в начальное состояние (позиции и скорости), заданное при его
инициализации
//!
//! @param object  Структура с информацией об объекте. Используется поле init.
//-----------------------------------------------------------------------------------------

void ResetObject (GameObject* object)
    {
    object->cur = object->init;

    object->animation = 0;
    }

//-----------------------------------------------------------------------------------------
//! Рисование игрового объекта
//!
//! @param object  Структура с информацией об объекте
//!
//! @note Для отладки сначала на месте объекта рисуется bounding box, потом копируется
//!       картинка. Если картинка не отобразилась, то bounding box будет виден.
//-----------------------------------------------------------------------------------------

void DrawObject (GameObject* object)
    {
    int frameSizeX = object->sizeX / object->numFrames;
    int frameSizeY = object->sizeY / object->numAnimations;

    int xSource    = object->frame     * frameSizeX;
    int ySource    = object->animation * frameSizeY;

    object->frame  = (object->frame + 1) % object->numFrames;

    if (Debug)
        {
        txSetColor (TX_PINK, 2);
        txSetFillColor (TX_NULL);
        txRectangle (object->cur.x - frameSizeX/2, object->cur.y - frameSizeY/2,
                     object->cur.x + frameSizeX/2, object->cur.y + frameSizeY/2);
```

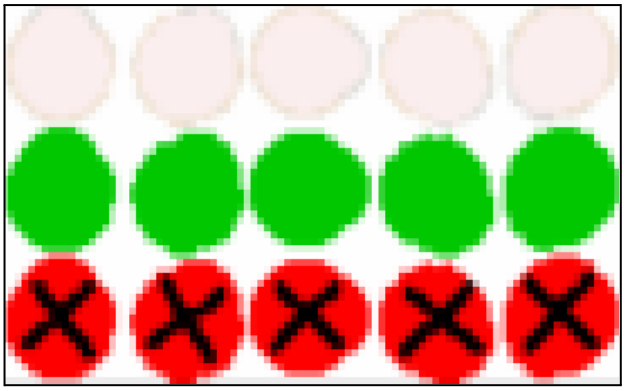
```
        txLine      (object->cur.x - frameSizeX/2, object->cur.y - frameSizeY/2,
                     object->cur.x + frameSizeX/2, object->cur.y + frameSizeY/2);
        txLine      (object->cur.x - frameSizeX/2, object->cur.y + frameSizeY/2,
                     object->cur.x + frameSizeX/2, object->cur.y - frameSizeY/2);
        }

    txTransparentBlt (txDC(),

                      object->cur.x - frameSizeX/2,
                      object->cur.y - frameSizeY/2,

                      frameSizeX,
                      frameSizeY,

                      object->image,

                      xSource,
                      ySource,

                      TX_WHITE);
    }

//-----------------------------------------------------------------------------------------
//! Перемещение объекта по базовым законам физики
//!
//! @param object  Структура с информацией об объекте
//! @param dt      Интервал времени движения
//!
//! @note Отталкивания нет, только равномерное/равноускоренное движение
//-----------------------------------------------------------------------------------------

void BaseMoveObject (GameObject* object, double dt)
    {
    object->cur.x += object->cur.vx * dt;
    object->cur.y += object->cur.vy * dt;
    }

//-----------------------------------------------------------------------------------------
//! Переустановка объектов в начальное состояние (позиции и скорости), заданное при их
инициализации
//!
//! @param objects     Массив структур с информацией об объектах. Используется их поля init.
//! @param numObjects  Количество объектов
//-----------------------------------------------------------------------------------------

void ResetObjects (GameObject objects[], int numObjects)
    {
    ResetObject (&objects[0]);

    for (int i = 1; i < numObjects; i++)
        {
        Point init = objects[i].init;
        objects[i] = objects[0];
        objects[i].init = init;

        ResetObject (&objects[i]);
```

```
        }
    }

//-----------------------------------------------------------------------------------------
//! Рисование массива объектов
//!
//! @param objects     Массив структур с информацией об объектах
//! @param numObjects  Количество объектов
//-----------------------------------------------------------------------------------------

void DrawObjects (GameObject objects[], int numObjects)
    {
    for (int i = 0; i < numObjects; i++)
        DrawObject (&objects[i]);
    }

//-----------------------------------------------------------------------------------------
//! Перемещение объектов по базовым законам физики
//!
//! @param object      Массив структур с информацией об объектах
//! @param numObjects  Количество объектов
//! @param dt          Интервал времени движения
//!
//! @note Отталкивания нет, только равномерное/равноускоренное движение
//-----------------------------------------------------------------------------------------

void BaseMoveObjects (GameObject objects[], int numObjects, double dt)
    {
    for (int i = 0; i < numObjects; i++)
        BaseMoveObject (&objects[i], dt);
    }

//-----------------------------------------------------------------------------------------
//! Вычисление расстояния между объектами, точнее, их состояниями.
//!
//! @param a  1-й объект
//! @param b  2-й объект
//!
//! @returns Расстояние между точками
//-----------------------------------------------------------------------------------------

double Distance (Point a, Point b)
    {
    return sqrt ((a.x - b.x) * (a.x - b.x) +
                 (a.y - b.y) * (a.y - b.y));
    }
```

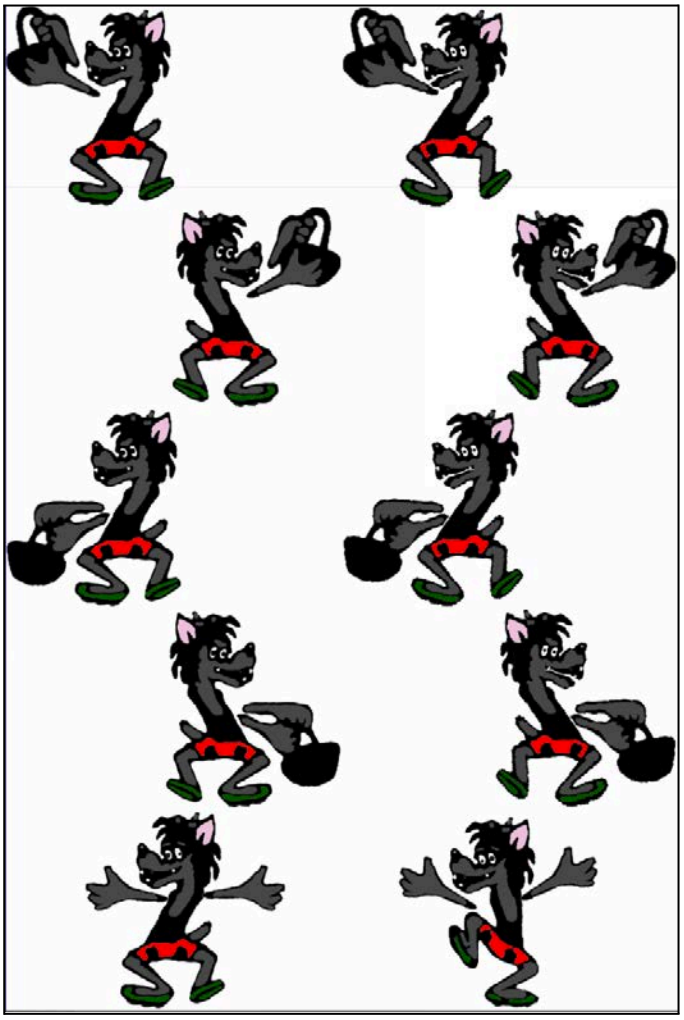
Картинка с фоном (Back.bmp):

Картинка с одной анимацией куриц (Chicken.bmp), присмотритесь, там кадры анимации чуть-чуть отличаются. Также картинка с катящимся яйцом (Egg.bmp), с тремя анимациями в нем:





Картинка с пятью анимациями волка (Wolf.bmp). Первые четыре используются для ловли яиц, при этом номер анимации соответствует номеру курицы. Последняя (снизу) – для танца волка в случае выигрыша:



Файл Doxyfile для генерации документации:

```
PROJECT_NAME     = "Wolf game"
INPUT            = WolfGame.cpp TXGameEngine.cpp
INPUT_ENCODING   = CP1251
OUTPUT_LANGUAGE = Russian
EXTRACT_ALL      = YES
CALL_GRAPH       = YES
HAVE_DOT         = YES
INLINE_SOURCES   = YES
SOURCE_BROWSER   = YES
IMAGE_PATH       = .
```

Фрагменты документации, которая получается не хуже, чем в TXLib'е:

Wolf game

Титульная страница Классы Файлы Поиск

Wolf game Документация

Игра про волка, который ловит яйца. Пример игры на библиотеке TXLib в учебном пособии "Программирование как проектирование".

Автор

Ilya Dedinsky

Дата

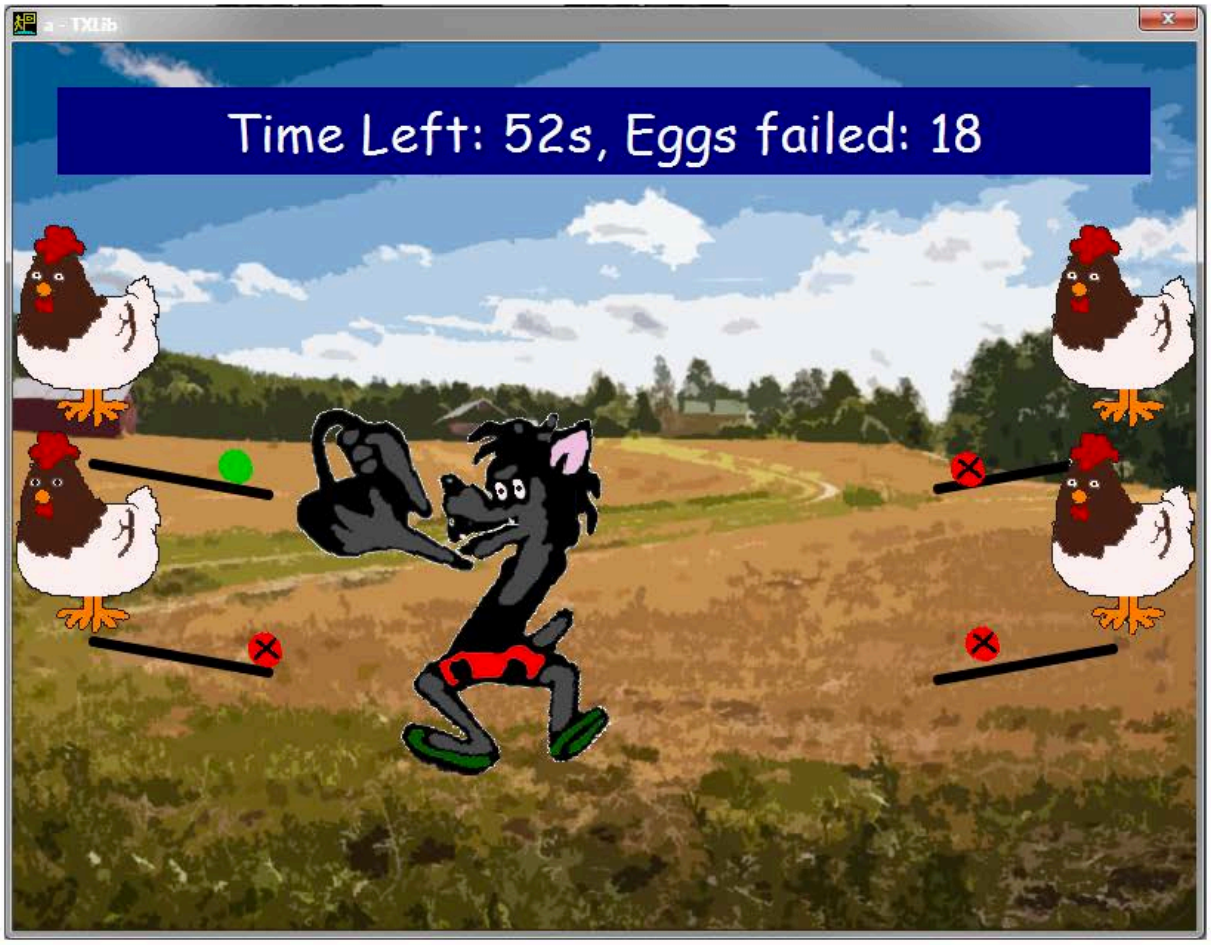
2023

Управление:

- Стрелка влево - Волк ловит левое верхнее яйцо
- Стрелка вправо - Волк ловит верхнее правое яйцо
- Клавиша End - Волк ловит левое нижнее яйцо
- Клавиша PgDn - Волк ловит правое нижнее яйцо
- Стрелка вверх - Волк переключается на ловлю верхних яиц
- Стрелка вниз - Волк переключается на ловлю нижних яиц
- Esc - Выход из игры

Файлы:

- **WolfGame.cpp** - Исходный текст игры
- **TXGameEngine.cpp** - Игоровой недодвижок



Создано системой doxygen 1.9.1

Wolf game

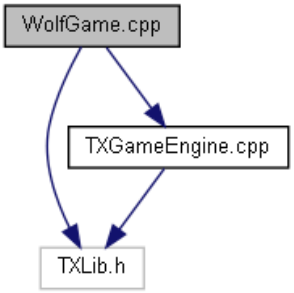
Титульная страница | Классы | Файлы | Поиск

Перечисления | Функции | Переменные

Файл WolfGame.cpp

```
#include "TXLib.h"
#include "TXGameEngine.cpp"
```

Граф включаемых заголовочных файлов для WolfGame.cpp:



См. исходные тексты.

Перечисления

enum **WolfAnimations** {
WOLF_LEFT_UP = 0 , **WOLF_RIGHT_UP** = 1 , **WOLF_LEFT_DOWN** = 2 , **WOLF_RIGHT_DOWN** = 3 ,
WOLF_DANCE = 4
}
Анимации волка Подробнее...

enum **EggAnimations** { **EGG_GOING** = 0 , **EGG_OK** = 1 , **EGG_FAILED** = 2 }
Анимации яйца Подробнее...

Функции

void **Play** ()
Функция главного цикла игры Подробнее...

void **DoDance** (**GameObject** *wolf, HDC back)
Функция танца волка. Подробнее...

void **DrawScore** (int failed, int timeLeft)
Отображение табло игры Подробнее...

void **DrawBanner** (COLORREF color, const char *text)
Рисование произвольной надписи на табло игры Подробнее...

void **ControlWolf** (**GameObject** *wolf)
Функция управления волком. Подробнее...

int **CheckEgg** (**GameObject** *egg, int chicken, **GameObject** *wolf, double dt)
Функция проверки одного яйца, снесенного курицей, на попадание в корзину Подробнее...

int **CheckEggs** (**GameObject** eggs[], int numEggs, **GameObject** *wolf, double dt)
Функция проверки всех яиц на попадание в корзину Подробнее...

void **RespawnEgg** (**GameObject** *egg, double dt)
Перезапуск яйца Подробнее...

Переменные

const bool	**Debug** = false
const int	**MaxTime** = 60 Максимальное время игры в секундах Подробнее...
const int	**MaxFailures** = 100 Максимальное количество разбитых яиц Подробнее...
const int	**NumChickens** = 4 Количество куриц Подробнее...
const int	**NumEggs** = 4 Количество одновременно катящихся яиц Подробнее...

Перечисления

◆ EggAnimations

enum **EggAnimations**

Анимации яйца

Элементы перечислений	
EGG_GOING	Яйцо катится
EGG_OK	Яйцо катится и сейчас попадет в корзину к волку
EGG_FAILED	Яйцо катится и сейчас разобьется

См. определение в файле **WolfGame.cpp** строка **55**

```
56      {
57      EGG_GOING       = 0,
58      EGG_OK          = 1,
59      EGG_FAILED      = 2
60      };
```

◆ WolfAnimations

enum **WolfAnimations**

Анимации волка

Элементы перечислений	
WOLF_LEFT_UP	Волк повернут влево, корзина вверху
WOLF_RIGHT_UP	Волк повернут вправо, корзина вверху
WOLF_LEFT_DOWN	Волк повернут влево, корзина внизу
WOLF_RIGHT_DOWN	Волк повернут вправо, корзина внизу
WOLF_DANCE	Волк танцует

См. определение в файле **WolfGame.cpp** строка **46**

```
47      {
48      WOLF_LEFT_UP    = 0,
49      WOLF_RIGHT_UP   = 1,
50      WOLF_LEFT_DOWN  = 2,
51      WOLF_RIGHT_DOWN = 3,
52      WOLF_DANCE      = 4
53      };
```

Функции

◆ CheckEgg()

```
int CheckEgg ( GameObject *  egg,
               int           chicken,
               GameObject *  wolf,
               double        dt
             )
```

Функция проверки одного яйца, снесенного курицей, на попадание в корзину

Если яйцо укатилось довольно далеко от точки появления, то яйцо либо попало в корзину, либо разбилось.

Если и номер курицы, которая его снесла, равен номеру анимации волка, то волк сейчас обращен к этой курице, и яйцо попало в корзину. Если не равен, то волк работает с другой курицей и проверяемое яйцо разбилось.

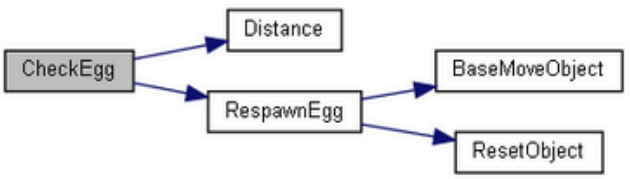
Аргументы

egg	Проверяемое яйцо
chicken	Номер курицы
wolf	Волк
dt	Шаг времени

См. определение в файле **WolfGame.cpp** строка **285**

```
286     {
287     double dist = Distance (egg->cur, egg->init);
288
289     if (100 <= dist && dist <= 140)
290         {
291         if (wolf->animation == chicken)
292             egg->animation = EGG_OK;
293         else
294             egg->animation = EGG_FAILED;
295         }
296
297     if (dist >= 140)
298         {
299         RespawnEgg (egg, dt);
300
301         if (wolf->animation == chicken)
302             {
303             if (Debug) { $s; printf ("Chicken %d: ", chicken); $g; printf ("OK\n"); }
304
305             return 0;
306             }
307         else
308             {
309             if (Debug) { $s; printf ("Chicken %d: ", chicken); $r; printf ("FAIL\n"); }
310
311             return 1;
312             }
313         }
314
315     return 0;
316     }
```

Граф вызовов:



2.15. Лабиринтные игры

Использование карт игр делает довольно легким написание игр-бродилок в лабиринте. На карте одним цветом можно отметить стены лабиринта, другими – места с сокровищами, ловушками, телепортами, выходами на другие уровни. Другие уровни реализуются просто заменой карты и фона, в целом алгоритм игры остается тот же.

При попадании в зону с сокровищами или в опасную зону можно переключаться на мини-игры, прохождение которых будет влиять на основную игру.

Такие игры легко преобразовать в интерактивные учебные пособия, например, для младших школьников, где мини-игры могут включать различные учебные задания – арифметические, логические, другие. Герой может собирать гербарий по ходу путешествия в лабиринте (и учить при этом ботанику пятого класса), перемещаться в "болоте" с "кочками", помеченными номерами, с определенными правилами перескакивания с кочки на кочку (например, текущий номер +N или -N, где N – целое число и т.п.). Может попадать в "реки", где к его координатам будет прибавляться скорость реки, и героя будет сносить течением и т.п. Набор идей здесь практически неисчерпаем. Ну и вашими пользователями могут стать ученики младших классов, если вы договоритесь с их учительницей. :)

В лабиринте могут находиться различные NPC, как правило, враги, столкновение с которыми чревато уроном. Поскольку поведение их более-менее однотипно, и их может быть достаточно много, разумно завести массив структур для хранения их данных. Для работы с этим массивом будут использоваться циклы, спрятанные в соответствующие функции для большей ясности. Таким образом, будет поддерживаться ясная и легко масштабируемая архитектура кода.

Переход с одного уровня на другой можно реализовать простой сменой фона и карты "на лету". При достижении точки выхода из уровня, мы уничтожаем HDC фона и карты с помощью txDeleteDC, сразу же загружаем новые фон и карту, и то, что нам вернула функция загрузки, присваиваем переменным, отвечающим за фон и карту:

```
if (...)                              // Если мы попали в зону смены уровня
    {
    txDeleteDC (back);                // Уничтожаем старый фон
    txDeleteDC (map);                 // Уничтожаем старую карту

    back = txLoadImage ("Back2.bmp");  // Загружаем новый фон
    map  = txLoadImage ("Map2.bmp");   // Загружаем новую карту
    }
```

Так как писать это напрямую в главном цикле игры приведет к грязному коду, лучше сделать функцию, отвечающую за смену уровня. Так как она будет менять значения переменных back и map, нужно будет передать в нее указатели на HDC, чтобы поменялись не копии, а оригиналы.

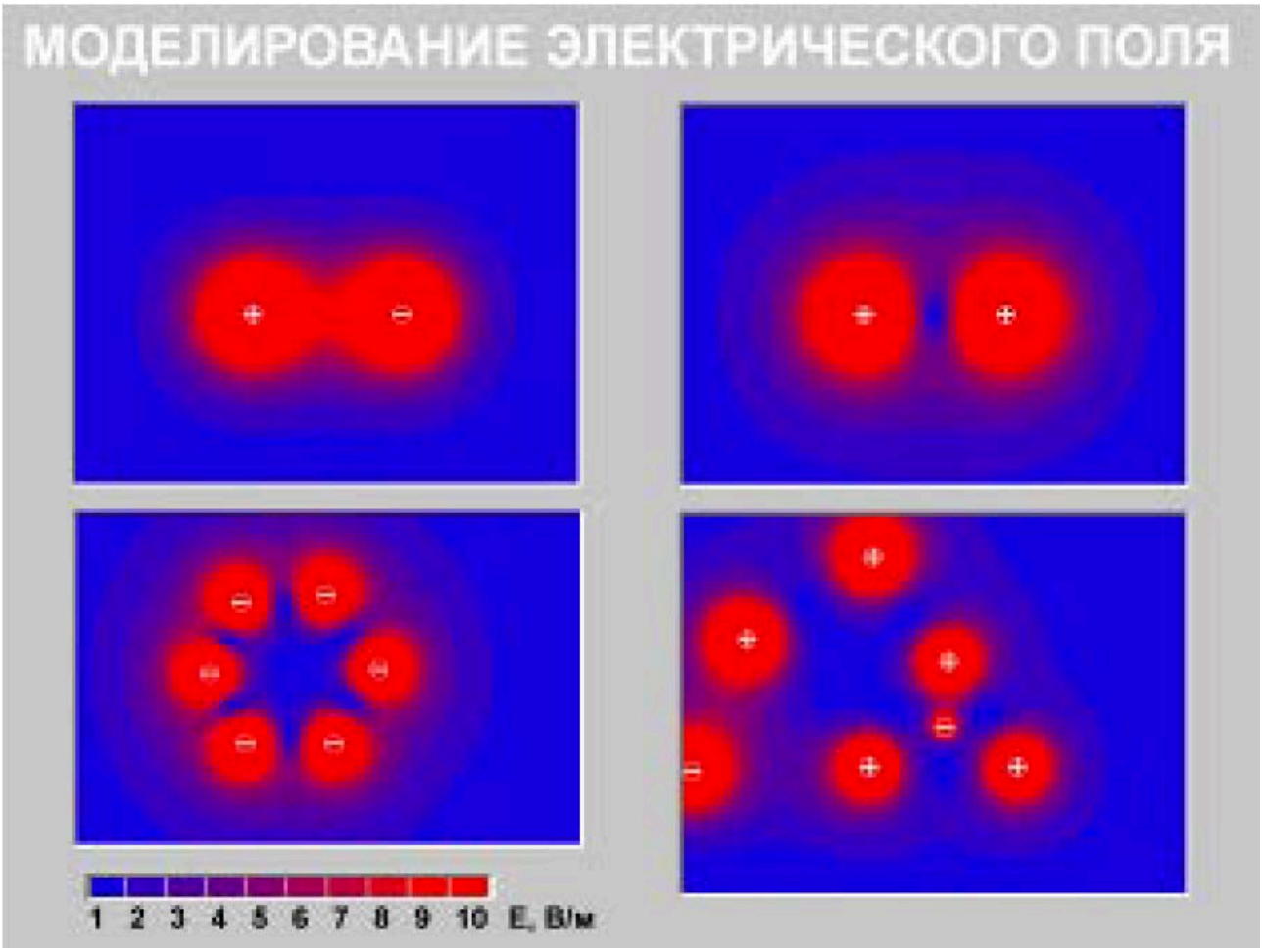
Проверка попадания в зону смены уровня, конечно, надо тоже реализовать в виде функции.

2.16. Моделирование физических явлений

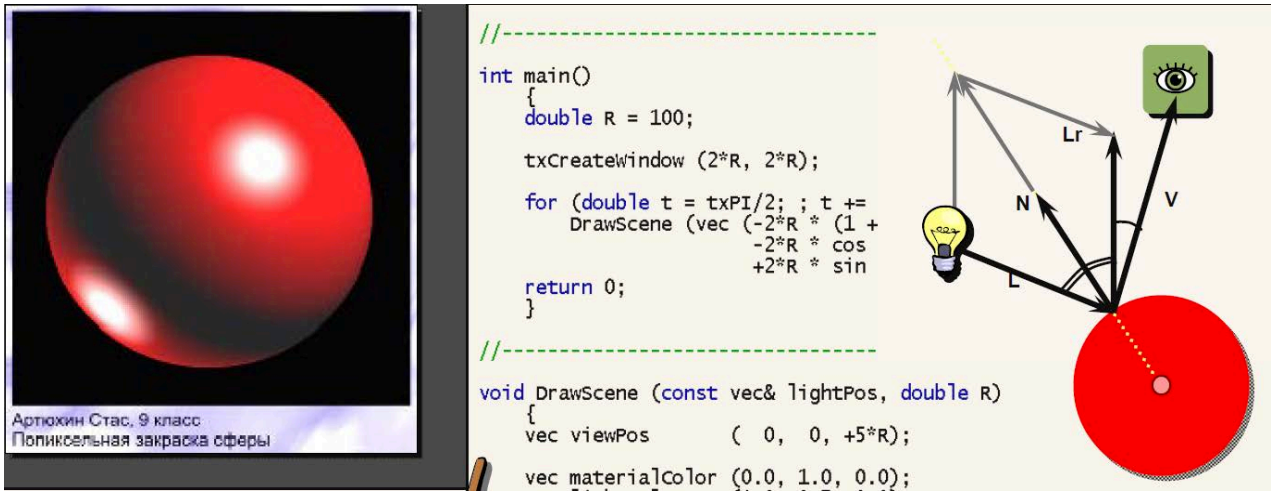
Применение программирования для моделирования явлений физики практически неисчерпаемо. Этим занимаются студенты, аспиранты, научные сотрудники, кафедры, лаборатории и институты. Иногда этим занимаются и школьники, и почему бы вам не

попробовать? Проще всего моделировать явления механические и молекулярно-кинетические, на базе механических. По сути мы с вами уже занимались моделированием физики, когда рассчитывали траектории движения объектов. В какой-то степени моделирование, не связанное с играми, проще, так как в таких моделях нет игровой логики, уровней и прочего, что характерно для игр. С другой стороны, зачастую требуется большая точность расчетов, иначе модель будет иметь мало смысла. Да и сил, действующих на частицы, больше, и сами силы хитрее.

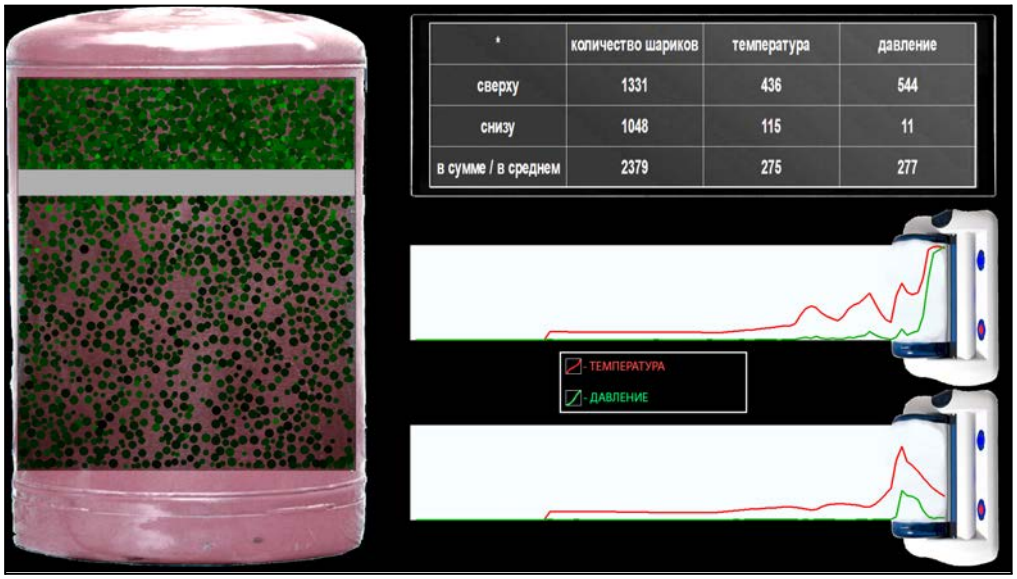
Учет различных сил дает много вариантов моделирования. Гравитационные силы – самые "естественные" для моделирования. Раньше уже упоминалась игра "посадка на планету", где мы практически держали в руках второй закон Ньютона, туда можно добавить и гравитацию. Достаточно полезная для физики игра "стрельба по замку из-за стены" (на эту тему есть много задач) затрагивает вопросы баллистики. Можно моделировать Солнечную систему, движение планет и комет в ней. Учет электрических взаимодействий приводит к моделированию движения заряженных частиц в электростатическом поле и другим задачам.



Можно строить модели освещения тел по закону Ламберта, эмпирически описывающего отражение света от поверхности. На рисунке ниже, правда, присутствуют блики, которые считаются не очень по-физически (по модели Фонга).

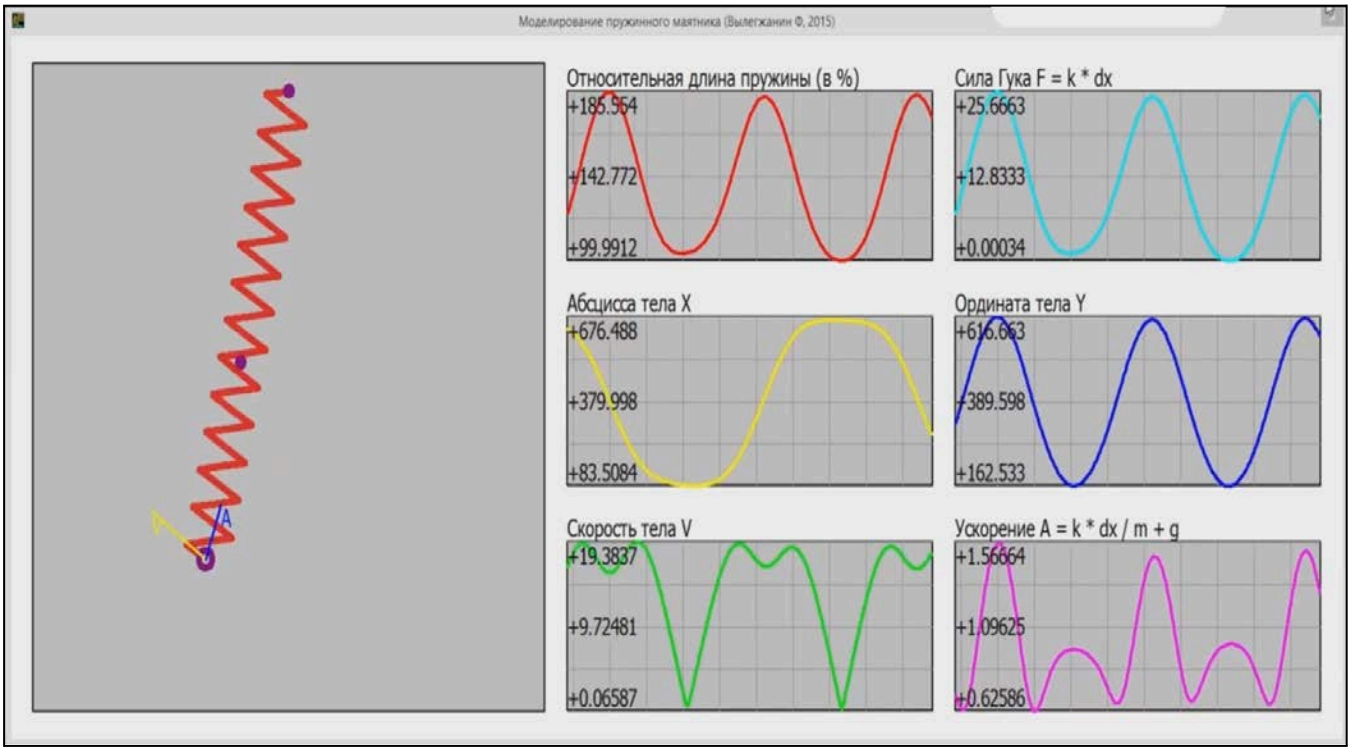


Одной из достаточно простых моделей может быть имитация поведения частиц идеального газа. Каждая частица (молекула) газа описывается структурой, структуры помещаются в массив, который обрабатывается циклами в функциях – все, как в игре. Для отображения физических параметров делаются функции, отображающие таблицы и графики. Вот один из примеров программ, написанных учеником 7 класса за 2 недели (но, правда, после года обучения проектным методом):



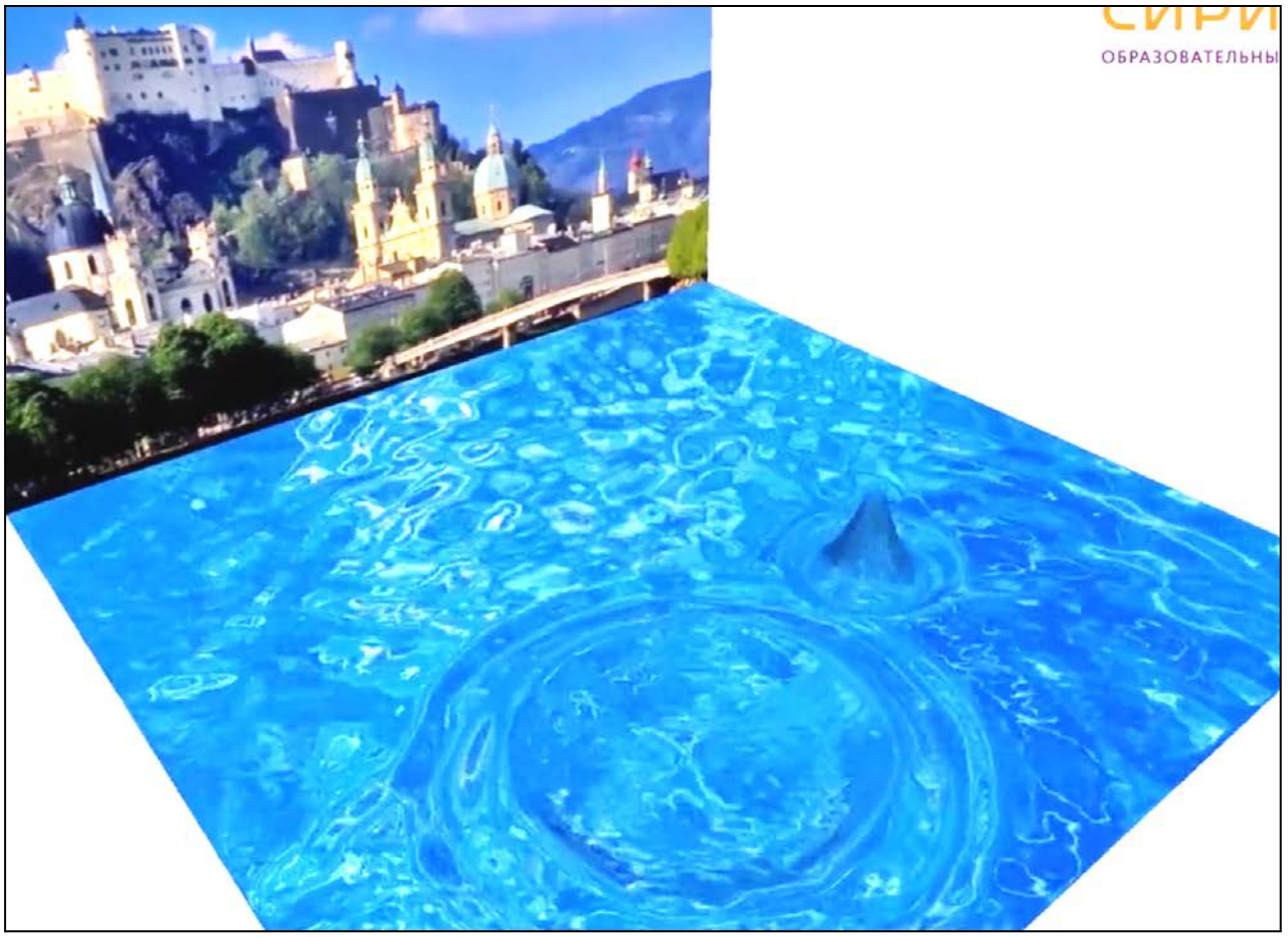
Программа моделирует поведение газа в сосуде с подвижной перегородкой. На картинке выше в верхнем отделении оказалось много быстрых (горячих) молекул, поэтому резервуар раскалился и покраснел. Температура вычисляется на основании средней кинетической энергии молекул, давление – среднему импульсу, передаваемому молекулами стенкам сосуда при столкновении с ними. Фактически все уравнения молекулярно-кинетической теории газов можно “пощупать” с помощью программы. Моделирование чуть более сложных газов (например, газа Ван-дер-Ваальса) уже является неплохим вопросом по выбору на экзамене по физике для студентов первого курса МФТИ.

Другая вполне посильная для реализации модель – движение так называемого апериодического маятника, груз которого закреплен на растяжимой нити или пружине. Программа моделирования в каждый момент времени рассчитывает все физические величины, рисует маятник и силы, действующие на груз, и выводит графики наиболее важных параметров системы.



Более сложные модели уровня 2-3 курса физико-математического вуза тоже доступны для реализации, но для них нужна помощь преподавателей, знающих такого рода явления и умеющих их рассказать “на пальцах”, так как строгое изложение теории все-таки находится на институтском уровне. Вот модель теплопереноса при нагреве твердых тел, реализованная школьником после двух лет обучения проектным методом. Изображение в центре экрана за докладчиком – не видео нагрева скрипичного ключа из проволоки, а строится программой:

Еще более сложная модель – поведение волн в жидкости согласно волновому уравнению, частному случаю уравнения Навье-Стокса. Здесь физики и математики еще больше, и, кроме того, автору программы пришлось отказаться от TXLib (о ужас!) по причине необходимости трехмерной графики. Сейчас TXLib в какой-то степени поддерживает ее, но, как вы понимаете, это все же учебная библиотека, миссия которой – стать ненужной, выполнив свою задачу и поспособствовав переходу к более сложным программным инструментам.



В моделировании физики очень важен контроль достоверности результатов, потому что при большом времени моделирования могут накапливаться значительные погрешности. Для этого часто применяют слежение за полной энергией системы (суммой потенциальной и кинетических энергий всех ее частей). При правильном моделировании полная энергия должна в среднем сохраняться, фактически она колеблется около некоторого константного значения. Если энергия самопроизвольно увеличивается или уменьшается, это означает, что параметры системы рассчитываются со значительной погрешностью. В общем случае подбор вычислительной схемы очень непрост, и этим занимается специальная наука – вычислительная математика. Однако часто можно подобрать такую модель, которая достаточно точно считается без применения сложных вычислительных хитростей.

Материалы этих работ, выполненных школьниками под руководством автора, можно найти по ссылке http://ded32.net.ru/news/2015-09-02-77.

Вместо заключения

Важно ли, какие игры и модели вы пишете? Нет. Совсем нет. Пишите любые, те, в которых вы можете осознать, как делать физику, геометрию, логику и графику. Главное, за чем вы, ваши друзья, коллеги, преподаватели и знакомые программисты должны следить – это качество кода, его ясность, структурность и масштабируемость. Внутри программа должна быть красивой, и этого, на самом деле, достаточно непросто достичь. У кого-то "уплывают в никуда" документирующие комментарии, у кого-то слабо разделение на файлы, а у кого-то оно слишком сложное; у кого-то беда с именами переменных, с "магическими числами" в коде, не замененными на константы; у кого-то десятки переменных вместо структур; у кого-то функции по сто строк, а у кого-то по тысяче, ох. Разработка программы – это искусство, а в нем много мелочей. Надо, чтобы у вас были надежные рефлексы, срабатывающие каждый раз, когда нужно правильно назвать переменную, вынести код в новую функцию и задокументировать его, собрать логически связанные переменные в структуры, сделать новую библиотеку, и каждый раз коммитить это в репозиторий и на сайт с ясными именами коммитов. Кажется, что это несложно, но это так не хватает многим начинающим разработчикам, и это очень частая причина их неуспеха на работе, вплоть до увольнения. И это то, что будет пригождаться вам в любом проекте, на любом языке, на любой платформе. Есть мнение, что для профессионального уровня нужно заниматься каким-то делом 10 тысяч часов – но мало кто уточняет, что считаются только часы, проведенные правильно. Для программиста "правильно" – это ясность, модульность, масштабируемость, надежность, эффективность, стандартность, переносимость, именно в таком порядке. Первые четыре свойства самые важные, без них бессмысленно развитие любого проекта. Начинайте развивать их в своих проектах и в себе уже сейчас, и вы уже сделаете первые шаги в правильном направлении – к своей профессии.