ЭВОЛЮЦИЯ ИГРЫ PunkMania

Артем Пимкин, 9 класс
<vk.com/nexx0f>

Введение

В этой статье я покажу эволюцию кода программы (по-научному – рефакторинг) на примере небольшой популярной в определенных кругах, хоть и достаточно давней моей игры PunkMania. Здесь я расскажу про часто встречающиеся ошибки в ходе разработки собственной игры (да и любой программы, не обязательно игры).

Для начала – пару слов о внутреннем устройстве этой статьи. В ней описываются различные стадии разработки игры, и каждому этапу посвящен отдельный раздел. Почему стадии разработки именно такие и почему они так называются, вы можете узнать из этой статьи, а также из моей презентации, найти которую можно на сайте ded32.net.ru в разделе «Конференция 2010 года». В примерах зеленым цветом будут выделяться законченные примеры хорошего кода – то, что я в результате оставил в финальном варианте своей программы. Красным цветом будет выделяться проблемный (существенно нуждающийся в доработке) код, о доработке которого нужно читать в этой статье далее. выделяться Участки кода, которые никак не изменялись по ходу развития программы, помечены синим цветом – они, как я посчитал, были неплохи с самого начала.

Этап прототипа

Первой появилась идея. Идея написать игру «что-то вроде Марио». Для начала необходимо было написать основу игры – прототип. Именно на нем будут основываться дальнейшие изменения и добавления. Эта версия лежит в папке «`Prototype version`».

На этой стадии у игры не было пока четкой игровой логики, было лишь некоторое представление о том, что должен «уметь» персонаж и как примерно должна проходить игра. Эти представления надо было протестировать, и в этом и состояла цель прототипа. Кроме того, чтобы не писать потом одно и то же много раз, уже на этой стадии я решил отделить код собственно игры от игрового движка, который я собирался использовать в последующем.

По логике вещей в игре должен быть главный персонаж, и он должен совершать некоторые действия, например: рисоваться на экране, контролироваться с клавиатуры. Было бы естественно, чтобы за них отвечали соответствующие функции (`Draw, Control`). Кроме того, персонаж всегда находится в каком-то месте уровня, а значит должен иметь свои координаты и скорости, выраженные в виде переменных (`x` и `y`, `vx` и `vy`). В результате получаем некоторый объект главного персонажа, в которую входит набор функций и набор переменных. Самым логичным шагом было бы создать класс, который содержит в себе все это (`class Punk`), что я и сделал:

```
class Punk
    {
    public:
    Punk (double x, double y, double vx, double vy);

    void Draw();
    void Control();
    ...

    double  x,  y;
    double vx, vy;
    ...
    };
```

Так же я поступил и с объектом игрового уровня. Он должен был содержать списки объектов, с которым будет взаимодействовать персонаж. Долго думал, как реализовать в полы и стены, находящиеся в нем. Сначала я попробовал создавать под каждый элемент уровня переменную соответствующего класса (`Floor, Wall`), но потом подумал: «А что если вот этих полов будет не два, а пять? А если двадцать пять? Неужели придется под каждый пол создавать своими руками переменную?». Очевидным выходом, разумеется, стали массивы, ведь в них могут поместиться необходимое мне множество элементов, и не составит труда обработать их в цикле – к примеру, нарисовать. В результате у меня имелись массивы элементов, с которыми может взаимодействовать персонаж (полы, стены – массивы `floorsData` и `wallsData`) и функции, обрабатывающие эти массивы. Также у меня были характеристики уровня (количество полов, стен – переменные `numFloors, numWalls` и стартовые координаты персонажа – `startX, startY`). Я поступил с ними так же как с главным персонажем, поместив их в класс (`class Level`):

```
class Level
    {
    public:
    Level();
    ...

    void LoadLevel (const char* name);
    void ResetLevel();

    int  MoveLevel (Punk_t* activePunk);
    void DrawLevel ();

    void GetPunkCoords (Punk* activePunk);
    void PutPunkCoords (Punk* activePunk);

    Wall  wallsData  [MAX_LEVEL_SIZE];
    int   numWalls;

    Floor floorsData [MAX_LEVEL_SIZE];
    int   numFloors;

    double startX, startY;
    };
```

Этот класс также содержит функцию `LoadLevel`, которая заполняет списки элементов, читая их из файла, функцию `Play` – она вызывается каждый игровой цикл, выполняя взаимодействие всех элементов уровня с персонажем, функцию `Draw`, рисующую уровень, и функцию `Reset`, которая вызывается для рестарта уровня, например, когда игрок проиграл или нажал кнопку “`R`”. Есть также пара функций, которые просто сокращают количество кода в главном игровом цикле.

Потребовалось также доработать классы `Floor_t` и `Wall_t`, которые ранее содержали лишь переменные: для каждого из них был написан код, обрабатывающий соприкосновение персонажа с ними – это функции `CheckFloor` и `CheckWall`. Они вызываются в функции `Move` класса `Level` для каждого элемента уровня.

Весь описанный код размещался в файле библиотеки игрового движка `PunkEngine`, который я собирался развивать и далее.

После этого для тестирования получившегося движка был написан еще один файл – «`PunkMania.cpp`». Он содержал загрузку уровня и главный игровой цикл (в котором происходит управление персонажа и его взаимодействие с уровнем), а также служебную функцию для изменения скорости игры в зависимости от скорости компьютера (подробнее о том, как работает эта функция, вы можете посмотреть в исходном коде). Именно в этом файле можно увидеть то, как применяются описанные ранее классы, как проста и легка работа с ними.

```
    while (true)
        {
            char openingLevel [512] = "";
            sprintf (openingLevel, "Engine Levels\\Level%d.txt",
                     hasToBeOpened);

            level.Reset  ();
            level.Load   (openingLevel);

            SetGameSpeed (time, &speed, &difficultyLevel);
            SetPunkPlace (activePunk, level.startX, level.startY);

            while (true)
            {

                level.GetPunkCoords (activePunk);

                gameStatus = level.Move ();

                level.Draw ();

                level.PutPunkCoords (activePunk);

                if ((gameStatus == LOSE) || (gameStatus == WIN)) break;

                activePunk -> Control  ();
                activePunk -> Draw     ();

                if (GetAsyncKeyState ('Z')) { goToMainMenu++; break; }
                if (GetAsyncKeyState ('R')) { break; }

                txSleep (speed);

                txSetFillColor (RGB (230, 230, 230));
                txClear        ();
            }

            if (gameStatus == -1) { Lose (&goToMainMenu); }
            if (gameStatus ==  1) { Win  (&goToMainMenu); }
            if (goToMainMenu > 0) { break; }
        }
```

Этап разработки игровой логики (геймплея)

После некоторых раздумий и советов от потенциальных игроков (мне как геймдизайнеру) был выработан следующий концепт (конкретная идея игры): главный герой должен иметь цель дойти до выхода и по пути поджечь коробки, раскиданные по уровню. А мешают ему сделать это противники, которые ходят по заранее заданному пути, и соприкосновение персонажа с ними приведет к неизбежному проигрышу.

Теперь давайте посмотрим на наш уже готовый прототип и на разработанный концепт с точки зрения программиста. Я рассматривал много вариантов архитектуры для добавления зоны выхода на следующий уровень, противников и коробок, первым в голову пришло хранить в классе уровня массивы с координатами и скоростями копов:

```
class Level
    {
    ...
    int cops_x  [MAX_LEVEL_SIZE];
    int cops_y  [MAX_LEVEL_SIZE];
    int cops_vx [MAX_LEVEL_SIZE];
    ...
    };
```

Но при добавлении остальных элементов уровня (т.е коробок и зоны выхода) таким же способом, сразу началась путаница в работе со всеми массивами, так как их вдруг стало слишком много. После добавления класс выглядел таким образом.

```
class Level
    {
    ...
    int  cops_x       [MAX_LEVEL_SIZE];
    int  cops_y       [MAX_LEVEL_SIZE];
    int  cops_vx      [MAX_LEVEL_SIZE];

    int  boxes_x      [MAX_LEVEL_SIZE];
    int  boxes _y     [MAX_LEVEL_SIZE];
    bool boxes _modes [MAX_LEVEL_SIZE];

    int  exit_x, exit_y;
    ...
    };
```

После добавления всех этих массивов работа с классом `Level` стала слишком непонятной и сложной. После осознания этого было решено немного поменять архитектуру класса. Первой идеей было попытаться как-нибудь объединить все переменные для каждого элемента уровня в некоторое хранилище, чтобы упростить работу с ними. А второй идеей было то, что копы, контейнеры и зона выхода – тоже, по сути дела, такие же элементы уровня, как и полы со стенами. Так что мешает создать для них, так же как для полов и стен, по структуре, а в классе `Level` создать массивы этих структур? Таким образом, в классе уровня все это стало выглядеть вот так:

```
class Level
    {
    ...
    Guard     gData  [MAX_LEVEL_SIZE];
    Container cData  [MAX_LEVEL_SIZE];
    Exit      exit;
    ...
    };
```

Согласитесь, так гораздо лучше и читабельнее. Ну а код вызова отрисовки и обработки элементов уровня в таком случае будет выглядеть так:

```
int Level::MoveLevel ()
{
    for (int i = 0; i < coulp; i++)
        floorsData[i].CheckFloor (&nowPunkX, &nowPunkY,&nowPunkVY);
    for (int i = 0; i < guacol; i++)
    {
        if (gData [i].CheckGuard (nowPunkX, nowPunkY) == -1) return LOSE;
    }

    for (int i = 0; i < concol; i++)
        cData [i].CheckContainer (nowPunkX, nowPunkY, this, concol);

    if (exit.CheckExit (nowPunkX, nowPunkY, burned, concol) == -1) return WIN;

    for (int i = 0; i < coulw; i++)
        wallsData[i].CheckWall (&nowPunkX, &nowPunkY, &nowPunkVX);

    return NOTHING;
}

void Level::DrawLevel ()
{
    for (int i = 0; i < coulp; i++)
        floorsData [i].DrawFloor ();

    for (int i = 0; i < guacol; i++)
        gData [i].DrawGuard ();

    for (int i = 0; i < concol; i++)
        cData [i].DrawContainer ();

    exit.DrawExit ();

    for (int i = 0; i < coulw; i++)
        wallsData[i].DrawWall ();
}
```

На мой взгляд, получилось не так уж плохо, тем более по сравнению с тем, что могло бы получиться. Теперь стоит разобрать созданные классы подробнее.

Рассмотрим копа. Согласно законам игры, его главная обязанность – взаимодействовать с панком с помощью функции CheckGuard и не давать ему поджигать коробки. Рисоваться он может с помощью функции DrawGuard. Поскольку он находится на уровне и перемещается от некоторой точки до другой точки, в его классе должны содержаться его координаты (x и y), скорости (vx и vy) и границы его перемещения (leftG и rightG).

```
class Guard
{
     public:
     int x, y;
     int vx;
     int intervalLeft, intervalRight;

     Guard ()        {x = 0; y = 0; leftG = 0; rightG = 0; vx = -10;}

     void DrawGuard  ();
     int  CheckGuard (double x, double y);
};
```

Разделение кода на рисование (DrawGuard) и исполнение служебных обязанностей (CheckGuard) здесь особенно помогло, потому что по игровой логике коп немного сложнее, чем пол и стены. Кроме того, объем расчетов столкновения у него также больше, чем у других элементов уровня.

Коробка так же, как и остальные элементы, находится в некотором месте уровня, значит, в ее классе должны быть координаты (x и y). Она также может рисоваться и взаимодействовать с панком с помощью своих функций DrawContainer и CheckContainer. Коробка может находиться в двух состояниях – подожженном и не подожженном, и, значит, нужна еще одна переменная, отвечающая за это (mode).

```
class Container
{
     public:
     int  x, y, mode;

     bool CheckContainer (double x, double y, Level *burned, int concol);
     void DrawContainer  ();
};
```

Ну, и последний класс, который неплохо было бы рассмотреть, это зона выхода с уровня. Она не может поджигаться, прыгать, бегать – она вообще никак не двигается. Поэтому с ней все проще. У нее должна быть функция рисования (DrawExit), функция (CheckExit), координаты x и y – и все:

```
class Exit
{
     public:
     int  x, y;

     int CheckExit       (double x, double y, int burned, int concol);
     void DrawExit       ();
};
```

После создания этих объектов, надо сделать то же самое, что и с остальными элементами уровня: создать массивы этих структур, которые будут обрабатывать функции MoveLevel и DrawLevel класса Level.

Подробный код этих функций вы можете увидеть в файле “Gameplay version\\PunkEngine1.0.cpp”.

Этап разработки и реализации дизайна

Итак, прототип движка и рабочая версия игры были написаны, играть в нее уже было достаточно интересно, но что же было делать дальше? Оказалось, что сделать надо было еще много. Игру сильно отличала от той же «Марио» неказистая внешность. Она выглядела как рисунок детсадовца, хотя в идеале, конечно, так выглядеть была не должна. Значит, нужно было добавить в мою игру красок, картинок и анимации (как говорят разработчики игр – арта). Сказано – сделано. Я попросил умевших рисовать друзей нарисовать мне картинки для игры в стилистике панков, так как ей я был намерен придерживаться. После того как они были нарисованы, встал вопрос как их встраивать в игру. Ведь с точки зрения дизайнера все было просто: главное нарисовать картинки, а потом они «сами как-нибудь в игру вставятся».

После этого я окинул ситуацию взглядом программиста. У меня есть несколько картинок, их надо загрузить в программу и как-то в ней использовать. Вроде просто, но появился вопрос: «где эти картинки хранить?». Самым очевидным решением этой проблемы кажется – в каждом классе (объекта, персонажа) создать переменную, отвечающую за картинку, и при создании объекта загружать эту картинку из файла. Да, это, несомненно, хороший вариант, но при этом у меня не будет централизованного доступа к этим картинкам, то есть если мне в какой-то части программы потребуются все картинки сразу, то мне придется вытаскивать их из нескольких объектов разных классов, что, согласитесь, неудобно. Поразмыслив над этим, я пришел к решению, что было бы неплохо создать некоторое объединение всех картинок моей игры, и каждый класс сам бы брал из этого объединения нужные ему картинки, а не загружал бы их сам. Итогом этих раздумий стал появившийся в моей программе класс `Images`, который при запуске программы загружает все необходимые картинки. Когда игровые объекты создаются, они берут свои, уже загруженные, картинки из объекта класса `Images`.

```
class Images
{
    public:
    HDC floor;
    HDC wall;
    HDC policeRight;
    HDC policeLeft;
    HDC burnedBox;
    HDC box;
    HDC darkPunkRight, darkPunkLeft, darkPunkStand;
    HDC sportPunkRight, sportPunkLeft, sportPunkStand;
    HDC funnyPunkRight, funnyPunkLeft, funnyPunkStand;
    HDC glamourPunkRight, glamourPunkLeft, glamourPunkStand;

    public:
    Images ();
};
```

Тогда, например, конструктор класса `Wall` до создания класса `Images` работал так:

```
Wall:: Wall (...)
{    ...
    pol = txLoadImage (“Images\\Wall.bmp”);
    ...
}

void Wall::DrawWall ()
{
   ...
   txBitBlt (x, y, ..., pol, ...);
    ...
}
```

При этом одна и та же картинка загружалась много раз. И еще, если я собирался поменять путь загрузки, мне это приходилось делать во всех конструкторах всех объектов, что, согласитесь, неудобно.

Но с появлением класса `Images` конструкторы просто берут уже загруженную картинку из объекта класса `Images`. А используется она так же, как и раньше:

```
void Wall::DrawWall(HDC wall) //Загружена один раз, хранится в классе Images
    {
    ...
    txBitBlt (x, y, ..., wall, ...); // Картинка рисуется, как обычно
    ...
    }
```

Согласитесь, второй вариант хранения картинки выглядит гораздо красивее и читается понятнее, кроме того он логически гораздо правильнее. Есть еще множество ситуаций, в которых использование нашего хранилища картинок (и не только картинок, а еще и звуков, скриптов, моделей – в общем случае, ресурсов) будет очень удобно. Такие проблемы часто возникают при хранении любых данных. В программировании игр такие хранилища известны как менеджеры ресурсов.

На этом этапе стоит закончить дописывание и переписывание файла «`PunkEngine1.0.cpp`». До полноценной программы остался лишь один шаг.

Финальный этап

Этим шагом является вся та оформительская рутина, которая должна быть в каждой игре. Она включает в себя создание всевозможных заставок, меню, титров и т.д. Ирония состоит в том, что на эту рутину уходит часто больше времени, чем на разработку самой игры.

С точки зрения гейм-дизайнера меню – это такая часть игры, в которой пользователь может не только запустить игровой процесс, но и настроить игру под себя, выбрать ее сложность, уровень и т.д. Задача же художника состоит в том, чтобы подать это меню максимально удобно для пользователя. Но и о том, что все должно выглядеть красиво, не нужно забывать. Максимальное удобство означает, что цвета не должны резать глаза, игроку должно быть сразу понятно, за что отвечает каждая надпись, все шрифты должны быть читабельны и сочетаться между собой. Кнопки в меню должны быть расположены интуитивно понятным способом. То есть моя задача как художника, была подобрать для моего меню такие картинки, такие кнопки и надписи, чтобы пользователю было с одного взгляда понятно, за что отвечает и к каким последствиям каждое его нажатие приведет.

С программистской точки зрение меню – штука простая, но достаточно скучная. Ведь что такое по своей сути меню, это несколько экранов (с настройками, кнопками и т.д), переход между которыми осуществляется за счет кнопок, нажимаемых пользователем. То есть из недостающих инструментов у нас отсутствует только кнопка. Так что мешает нам сделать ее? Ведь кнопка – это, фактически, красивая обертка над оператором `if`, проверяющем нажатие. Сначала я написал это напрямую и получил примерно такой код:

```
int Menu (Punks* myPunks, Punk** activePunk, HDC logo)
    {
    ...
    if ((txMouseX () >= but1X1) && (txMouseX () <= but1X2) &&
        (txMouseY () >= but1Y1) && (txMouseY () <= but1Y2) &&
        (GetAsyncKeyState (VK_LBUTTON)))
        DoMenu1();

    ...

    if ((txMouseX () >= but9X1) && (txMouseX () <= but9X2) &&
        (txMouseY () >= but9Y1) && (txMouseY () <= but9Y2) &&
        (GetAsyncKeyState (VK_LBUTTON)))
        DoMenu9();
    ...
    }
```

При этом исходный код функции `DoMenu9` выглядел вот так:

```
void DoMenu9 ()
    {
    ...
    if ((txMouseX () >= button33X1) && (txMouseX () <= button33X2) &&
        (txMouseY () >= button33Y1) && (txMouseY () <= button33Y2) &&
        (GetAsyncKeyState (VK_LBUTTON)))
        DoSubMenu33();

    ...

    if ((txMouseX () >= button51X1) && (txMouseX () <= button51X2) &&
        (txMouseY () >= button51Y1) && (txMouseY () <= button51Y2) &&
        (GetAsyncKeyState (VK_LBUTTON)))
        DoSubMenu51();  // O_o
    ...
    }
```

Не кажется ли вам, что в этих функциях есть не просто повторяющийся элемент, а ПЯТЬДЕСЯТ ОДИН повторяющийся элемент? То есть для того чтобы проверить на нажатие находящуюся на экране 51 кнопку нужно достаточно много кода, и ведь это только код проверки на нажатие, а кнопки надо еще и рисовать на экране! Но если количество кода не кажется вам внушительным, то представьте, сколько переменных надо объявить. Если кнопка задается четырьмя числами, то на 51 кнопку необходимо 4*51 = 204 переменные. Вы действительно в состоянии прямо в коде программы, своими руками объявлять 204 переменные? Если да (0_o), то представьте, что было бы, если бы кнопок было 501? Согласитесь, возникает вполне естественное желание убрать код проверки на нажатие как минимум в функцию – что я первым же делом и сделал. Но, к сожалению, необходимость объявлять собственноручно 204 переменные осталась. Возник вопрос, как можно самым простым образом объединить переменные с этой самой функцией проверки на нажатие и отрисовку? Конечно же, помогли не раз спасавшие меня классы. Результатом стал класс (или структура) кнопки с находящейся в ней текстом. Это был класс TextButton:

```
class TextButton
{
    public:
    int         x1, y1, x2, y2;
    int         nx, ny;
    const char* text;

    TextButton  (int nx1, int ny1, int nx2, int ny2,
                 int tx, int ty,
                 const char* newText);
    int Action  ();
};
```

По сути, класс любой кнопки – это класс, который умеет рисовать кнопку и проверять, не нажата ли она мышью. Проверка на нажатие – это тест кнопок мыши и два двойных неравенства, проверяющих нахождение точки (где находится курсор мыши) внутри прямоугольника (который является кнопкой). Для этого класс содержит координаты кнопки, текст или картинку, или и то и другое вместе.

Я думаю, рассказ о кнопке на этом стоит закончить, и перейти к рассказу о, собственно, меню. Как я уже раньше говорил, меню – это некоторая система экранов, связанных между собой переходами через нажатие клавиши. Самым очевидным решением было бы просто напрямую написать код всех экранов в одну функцию и связать их с помощью оператора if...

...Что я первым делом и сделал, засунув весь код описывающий меню, в одну функцию. Что получилось (утрированно) вы можете увидеть ниже.

```
int Menu_t::Process()
    {
    ...
    if (optionsButton.Action() == PRESSED)
        {
        while (true)
            {
            ...
            if (SubOptons.Action() == PRESSED)
                {
                ...
                while (true)
                    {
                    ...
                    if (SubSubOptios.Action() == PRESSED) ...
                    ...
                    }
                ...
                }
            ...
            }
        }
    ...
    }
```

Согласитесь, что выглядит нечитаемо. А ведь это всего лишь подменю двойной вложенности. Пока мне в голову не пришло ничего лучше, чем создать под каждое подменю отдельную функцию, для сокращения кода в основной функции меню и увеличения его читабельности. В результате получился примерно такой код:

```
Menu (…)
{
...
if (OptionsButton.Action()) DoOptions(…);
...
}

DoOptions (…)
{
...
if (PunkChooseButton.Action()) DoOptionsPunkChoose(…);
...
}

DoOptionsPunkChoose (…)
{
...
}
```

Заключение

В этой статье я хотел на примере несложной игры показать, какие проблемы могут возникнуть у вас при работе над аналогичным проектом, где может встать вопрос хранения данных и удобной работы с ними. Пути решения проблем, предложенные в этой статье, далеко не единственные, я лишь избрал их для этой игры, в тот момент, когда писал ее. Сейчас я выбрал бы уже более продвинутые решения, по завершении этого проекта они стали очевидными для меня (защиту данных, виртуальные функции, менеджеры объектов и многое другое). Но в седьмом классе, когда я писал эту игру, я не знал всего этого, и еще мне не всегда было понятно, чем лучше один способ и чем хуже другой. Поэтому я решил представить не тот код, который бы я сейчас написал (а переписал бы я практически все), а тот, который остался после моего седьмого класса, чтобы тем, кто начинает писать, в чем-то было бы легче.

Здесь я привел краткий перечень проблем и выбранных мной решений:

Проблемы	**Решения**
Отсутствие удобного управления главным персонажем, необходимость создавать много переменных каждый раз при создании нового персонажа.	Создание класса `Punk`, который является объединением всех функций и переменных, связанных с главным персонажем.
Скопление массивов переменных, отвечающих за различные элементы уровня, в классе `Level`. Крайнее неудобность работы с этими массивами и большая вероятность в них запутаться.	Создание структур для каждого элемента уровня и функций работы с этими структурами. После этого, задание содержимого уровня в классе `Level` через массивы этих структур.
Огромное количество одинаковых картинок, хранящихся в памяти программы, и сложность получения необходимой картинки в нужный момент.	Создание менеджера картинок (класса `Images`), в котором хранятся все необходимые картинки. В таком случае получаем быстрый и удобный доступ к любой картинке, появляется возможность с легкостью передать нужную картинку в нужную функцию.
Много похожих друг на друга кусков кода, обрабатывающих нажатие на кнопки в меню. Большой размер функции `Menu` относительно остальных функций.	Создание класса `TextButton` для сокращения повторяющейся части кода. Разделение функции `Menu` на разные функции, вызывающие при необходимости друг друга.

Еще, на правах рекламы, хотелось бы сказать про то, что я создал вторую часть игры PunkMania. Она называется PunkMania – Dark Punk Days. Она уже гораздо сложнее, чем пример, рассмотренный в статье, но разрабатывалась она по той же схеме (Прототип – Геймплей – Дизайн – Меню), что еще раз доказывает, что эта схема лучше, чем «просто кодить». Скорее всего, вторая часть моей и игры в скором времени появится на сайте `ded32.ru`.