ТЕХНОЛОГИЧНО УЧИЛИЩЕ ЕЛЕКТРОННИ СИСТЕМИ

към ТЕХНИЧЕСКИ УНИВЕРСИТЕТ – СОФИЯ

Д И П Л О М Н А Р А Б О Т А

Тема: 3D Survival Horror игра

Дипломант: *Мартин Раловски*

Научен ръководител: *Панайот Йаназов*

С О Ф И Я

2 0 1 1

Увод

Развитието на видео игрите постоянно бележи напредък. Започвайки през 50-те години на 20-ти век, видео игрите се усъвършенстват постоянно, минавайки през класическите конзоли като *Nintendo Family*, *Atari 2600* и *Sega Mega Drive* със заглавия на игри като *Mortal Kombat*, *Super Mario*, *Street Fighter* и *Sonic*, достигайки до конзоли като *XBOX 360*, *PlayStation 1/2/3*, *Wii*, както и най-бързо развиващата се платформа за игри - персоналният компютър.

Компютърните видео игри, в момента, „изживяват“ своя Ренесанс, благодарение на бързото развитие на хардуерните компоненти. Техният напредък прави възможно достигането на нови висини в компютърната графика, играене на игри на няколко минитора с различни камери, както и на 3D монитори. Точно този напредък ни дава възможност да се насладим на игри като *Call Of Duty Modern Warfare 1 / 2*, *Left 4 Dead 1/ 2*, *Need For Speed Most Wanted*, *Crysis*, *Half-Life 2*, *Sacred 2*, *Command & Conquer 3* и много други.

В началото единственият жанр е бил аркадната игра, предлагаща уникално изживяване за своето време. Постепенно развитието на игрите бележи своя ръст и се появяват все повече и повече жанрове. При появата си, шахът, игран срещу машина, изпълва с възторг хората, докоснали се до него. Игри като *Tic-Tac-Toe* (морски шах) , предназначени за определен хардуер, са считани за основополагащи при видео игрите. В днешно време съществуват най-различни жанрове игри – симулатори, шутъри, екшън, ужаси, ролеви, както и различни поджанрове.

Целта на настоящата дипломна работа е да бъде създадена *Rest In Peace (R.I.P.)* - триизмерна компютърна игра от трето лице от жанр сървайвал хорър, с реалистична физика и възможност за игра в *single* и *multiplayer* в локална мрежа, както и създаване на редактор на нива за изграждане и подреждане на нивата в играта.

Глава 1 Игрови жанрове. Survival Horror. APIs и езици за разработка на игри.

1.1 Игрови жанрове

Игровите жанрове започват своето начало още с появата на игрите. Жанрът на една игра описва нейната същност и какъв ще бъде главният елемент в сюжета ѝ. С развитието на игрите са се развили и видовете жанрове, защото познатите до сега вече не са успявали да опишат даденият продукт достатъчно добре.

Игровите жанрове са много и могат да се комбинират помежду си – *Dead Space 2* (3D екшън, ужаси). Комбинацията между жанрове може да послужи като начин за по-добро описание на играта и по този начин може да привлече повече потребители, които биха искали да я пробват, защото ако една игра е 3D екшън и ужас тя може да привлече любители на 3D екшъна и такива на ужасите.

Жанрът е важна част от играта, защото по този начин я представя пред света с една дума.

1.2 Survival Horror[1]

1.2.1 История на хоръра

Историята на хоръра може да се проследи до началото на Великата Инквизицията, която поражада неописуем ужас в сърцата на хората. Множеството различни митологии често са използвани като основа за хорър сюжет. В митологията съществува тайнственото, паранормалното и нещо отвъд човешкото. Произходите на сървайвал хорър игрите могат да се проследят до ранната хорър фантастика. Архетипи са свързвани с книгите на Хауърд Филипс Лъвкрафт.

1.2.2 Характеристика на хоръра

Fear is the most powerful emotion in the human race and fear of the unknown is probably the most ancient. You're dealing with stuff that everybody has felt; from being little babies we're frightened of the dark, we're frightened of the unknown. If you're making a horror film you get to play with the audiences feelings. Така описва хоръра американският режисьор Джон Карпентър. За

[1] http://en.wikipedia.org/wiki/Survival_horror

разлика от другите жанрове, в ужасите се търси начин публиката да изпита страх. В изкуството на хорърите важен фактор е обстановката. Чрез нея може да се създаде най-различно усещане за ужас, мистерия или страх от непознатото и тъмното. Този жанр често е съпровождан от вулгарен език, голямо количество насилие, гняв и ситуации, предизвикващи страх.

Игрите от този тип са обикновено от трето лице. Главният персонаж е поставен в необичайна и враждебна нему обстановка: изоставен град, призрачна къща или замък. Тук той е принуден да се бори за оцеляването си срещу демони, чудовища, зомбита и друг подобни създания. Средствата му за защита в повечето случаи са скромни. Персонажът не притежава свръхестествени сили, мощни и разнообразни оръжия, а боеприпасите са необичайно малко. За сметка на това е нужно да проявява съобразителност, комбинирайки и използвайки различни предмети. Сюжета съдържа мистерия, която трябва да бъде разгадана в хода на играта. От особено значение е мрачната атмосфера, подсилвана със всички възможни средства, включително и музиката. Създателите на този род игри умишлено се стремят да внушат на играещия чувство на страх, объркаността и дори депресия.

1.2.3 3D Survival Horror

За основоположници на *3D Survival Horror*-а като жанр в компютърните игри се считат игрите *Resident Evil 1* (1996) и *Alone in the Dark* (1992). Преди тях, обаче, излиза играта *Alien: The Computer Game*, по едноименния филм на Джеймс Камерън. Играта, макар и ограничена от към възможности предоставя доста стряскащи и спиращи дъха моменти. В съвремнни игри като Dead Space може да се открият елемнти, лежащи върху тази игра. Въпреки това Resident Evil и Alone in the Dark за първи път показват възможностите на жанра хорър и го дефинират като такъв в гейм индустрията. *Resident Evil* за първи път определя жанра. Тя е считана за класика заради няколко нейни особености:

- Статичната камера, която отнема контрола от играча. Такава игра буди известни негативизми в сегашните геймъри, но камерата е компенсирана от добрата детайлност на обстановката.

- Обстановката – разработчиците успяват да постигнат удивително добра атмосфера с достатъчно много детайли, за да пресъздадът страшната среда.
- Сюжетът - сюжетът разказва за изгубените членове на специялният екип *S.T.A.R.S*. и останалите Крис Редфийлд и Джил Валнетин, които се опитват да ги намерят.
- Звукът – звукът в играта може да се мери с всяка една съвременна игра. В *Resident Evil* са успели да вдигнат музиката на много високо ниво и тя допринася за злокобната обстановка.
- Престрелките и преследванията – престрелките в играта са доста „инфарктни“, а преследванията в късите коридори могат да са много стряскащи, но и забавни.

След време излиза *Silent Hill* – игра, която прави хорърите по-различни – повлияна от японския хорър, сюжета набляга върху психологическият характер на ужаса. *Silent Hill* вкарва играча в атмосферата на град с непрестанна мъгла и изскачащи зад ъгъла зомбита.

В днешно време игри като *Left 4 Dead* 1 и 2, *Dead Space* 1 и 2, *F.E.A.R* са игри, представители на хорър жанра.

Left 4 Dead е игра, която поддържа два вида режима на игра – *Single Player* и *Multiplayer*. В режимът Single Player потребителят избира герой измежду четири възможни, като другите трима са управлявани от компютъра. В този режим играчът изиграва четири кампании, които имат по няколко различни нива.

Режимът *Multiplayer* дава възможност за игра между няколко човека, като може да се играе играчи срещу зомбита, управлявани от компютър, или играчи срещу играчи като на едно от нивата отборите се сменят като *Survival* или *Infected*.

1.3 APIs и езици за разработка на игри.

1.3.1 Програмни езици

Когато първоначалният дизайн на играта е определен, трябва да се избере езикът за разработка. Изборът зависи от много фактори, например запознатостта с езиците на разработчиците, платформите, за които ще бъде разработвана играта (като *PlayStation* или

Microsoft Windows), изискванията за скоростта на изпълнение и езика на използваните енджини, APIs или библиотеки.

Днес, понеже е обектно ориентиран и се компилира до машинен код (*native*-езикът за съответната платформа), най-популярният език за разработка на игри е *C++*. Въпреки това, *Java* и *C* са също популярни, но неподходящи за някои проекти. Асемблер е необходим за някои конзоли за игри и също когато алгоритмите трябва да са максимално възможно бързи или да има много малко забавяне. *C#*, *Ada* и *Python* са имали по-малко влияние на инустрията и са използвани главно от ентусиасти, за които разработката на игри е хоби. Все пак, напоследък *C#* е доста популярен за програмиране на инструменти за разработка на игри и дори използван от някои компании за комерсиални игри.

Скриптови езици от високо ниво все повече и повече се използват за конзоли в самите игри или за разширения към игри написани на език от ниско или средно ниво като *C++*. Много разработчици са създали специфични езици за собствените си игри. Например *QuakeC* на *id Software*, *UnrealScript* на *Epic Games*.

Vertex и *pixel* шейдъри също са набрали популярност заедно с широкото разпространените програмируемите *GPU*-та. Това довежда до използване на *High Level Shader* езици (като *HLSL* или *GLSL*) в програмирането на игри, но те все още не могат да се използват за цялата логика на играта.

1.3.2 APIs

Най-популярната платформа за игри на персонален компютър е *Windows*. Най-често използваното *API* за игри за *Windows* е *DirectX*. *OpenGL* също е възможен избор, но то се използва повече за приложения и други програми, отколкото за игри. *DirectX* е колекция от *APIs* за игри, състои се от *Direct3D*, *Direct2D*, *DirectPlay*, *DirectInput* и други, изпълняващи различни функции. *Direct3D* и *Direct2D* са графичните *APIs* съответно за *2D* и *3D* игри, *DirectPlay* се използва за създаване на multiplayer, а *DirectInput* за вход на данни от мишка, кавиатура, волан, джойстик, геймпад и други.

OpenGL е portable *API*. Използва се за разработка или портване на игри за *Linux* и *Mac* и също се поддържа от някои конзоли (*Nintendo*, *GameCube*, *NintendoDS*, *PSP* и други).

SlimDX и *OpenTK* са wrapper-и за *.NET* съответно на *DirectX* и *OpenGL*. *OpenTK* работи и с *Mono*.

Глава 2 Изисквания към играта и редактора на нива. Избор на средства за разработка. Основна структура на XNA и на играта.

2.1 Изисквания към играта и редактора на нива.

- Изисквания към играта:
  - Да се създаде триизмерна сървайвал хорър игра, която да може да се играе в *Single* и *Multiplayer*. Да има реалистична физика, звукова система, да има меню за графични и звукови опции на играта – да може да се променя резолюцията, силата звука, *VSync* и други, възможност за запазване на прогреса и настройките.
  - Главният герой се намира в типичната за сървайвал хорър игри атмосфера. Целта му е да убива зомбита, като за целта разполага с различни оръжия. Предвидените в началото са брадва, арбалет и револвер.
  - Целта на физичния енджин използван в играта е да се придаде допълнително ниво на реализъм, например при сблъсък на два или повече обекта те да реагират както биха реагирали в реалния свят.
  - Да има инсталатор, който да инсталира играта и необходимите библиотеки и framework и за нейното стартиране.
- Изиксвания към редактора на нива:
  - Редактора на нива е програма, с която се изгражда нивото или „света” в играта. Основната му функционалност трябва да включва възможност за транслации, ротации и скалиране на моделите по x, y и z, триене, копиране, както и определяне кои модели да бъдат третирани като физични, кои да са статични и кои не, определяне на началната позиция на героя и на враговете, както и областите, в които може и не може да се ходи.

- Редактора на нива да може да запазва нови, както и да отваря и редактира вече създадени нива.

- Той също трябва да зарежда терена на играта и да дава опция за директно поставяне на модела върху терена.

- Редактора се позлва само то разработващите играта, но не и от клиента.

2.2 История

Rest In Peace дава възможност на играча да изживее драмата и ужаса на човек, завърнал се в родното си имение в търсене на информация за своя прадядо, изчезнал безследно преди години.

В един мрачен ден, героят се сеща за своя прадядо и решава да тръгне по неговите следи. Така той се връща в родния си град и го намира изоставен и полуразрушен – останали са само няколко къщи, сред които е тази на дядо му, и църквата.

В началото той вижда хора, много приличащи на самия него. Когато се опитва да ги заговори – те го нападат. Така започва и битката за достигането до къщата. Избивайки дузина зомбита, той попада в къщата на своя прадядо. Истината трябва да бъде разкрита в кабинета, който, естествено, се намира на последния етаж на сградата. Докато стигне до там, играчът трябва да си проправи път през ордите от зомбита. Достигайки заветната цел той намира дневника на своя роднина и разбира, че дядо му е намерил машина, която може да копира хора, но са се получават странични ефекти. Така след като мистериозно изчезва, машина бълва стотици зомбита, които успяват да избият жителите на градчето. На последната страница той вижда бележка, на която пише, че единственият човек, с когото е говорил за това, е бил свещеникът в местната църква, който се е заклел да унищожи тази зараза. Откривайки тези факти той се отправя към църквата с надеждата да намери повече информация. Когато стига църквата, той разбира, че прадядо му няма да намери покой, докато не бъдат унищожени всички същества и машината, които е създал.

2.3 Избор на средства за разработка.

Средствата, с които е реализирана играта са езикът *C#* под *.NET Framework 4*, *XNA Framework 4* и *Jitter Physics* за физиката. За редактора на нива са използвани и *Wndows Forms*.

Главната причина за избора на *C#* е, че писането на код с него е много продуктивно. Предимства на езика като интерфейси, generics, делегати го правят по-чист от *C++*. Практики на разработка като unit тестове също са по-лесни на *C#*. Тази яснота води до по-четлив код и по-малко бъгове и по-лесна поддръжка.

Втората печалба на продуктивност идва от достъпа до *.NET Framework*. От *XML* манипулации за конфигурация, до вградените математически библиотеки и сокети. Много от основните функции, за които човек може да се сети са написани. Това освобождава от писането на код и оставя надеждни, тествани библиотеки да свършат работата, която не искаме да губим време да вършим. C# ни оставя да се концентрираме на самата игра, а не да пишем енджин или да учим енджина на някой друг. *Visual Studio* и *Visual Studio Express* имат чудесни инструменти за рефакториране поддържащи преименуване, извличане на методи и интерфейси. Също така разделянето на проекта на отделни *DLL* файлове е тривиално в *C#*. Това помага при поддръжка и енкапсулация на кода.

Jitter Physics е физичен енджин, изцяло написан на *C#*. Причините той да бъде използван, а не *binding*-и на някой енджин написан на *unmanaged* код като *Havok*, *BulletX* или *NVIDIA PhysX*, които със сигурност биха дали далеч по-добра производителност са две. Едната е, че *Jitter* е изцяло *portable* – може да работи на всички платформи, които поддържат *.NET*, и дори *Mono*. Другата причина е недостатък на *C#* - директното *link*-ване на *C++* (не-*COM*) библиотеки не е толкова тривиално, колкото *link*-ване на други *.NET* библиотеки. Освен това, *XnaHavok* и *PhysX.Net* са *beta* версии и все още не са напълно стабилни.

XNA(съкращение от *XNA is Not Acronym*) е *framework* за разработка на игри за *Windows*, *XBox 360*, *Zune* и *Windows Phone 7*. *XNA* използва модифицирана версия на *Common Language Runtime* (*CLR*), която е по-оптимизирана и подходяща за игри. Целта на *XNA* е да освободи програмистите на игри от писането на повтарящ се код и да обедини различни

аспекти от разработката на игри в една система. *XNA Framework* енкапсулира технологични детайли от ниско ниво свързани с програмирането на игри, така, че *framework*-а да се грижи за тях, което позволява на разработчиците на игри да се фокусират директно върху съдържанието и самата игра.

Всичко това прави *XNA* и *C#* перфектният избор при разработка на игри в екип от един или двама човека.

2.4 Основни класове в XNA

Всички игри следват един основен принцип на работа, който е същият или много подобен на този показан по-долу.

```
Initialize();
while( isRunning )
{
    Update();
    Draw();
}
Dispose();
```

XNA не е изключение. В `Initialize()` се инициализират или зареждат всички необходими променливи и ресурси. След това в цикъл се викат `Update()` и `Draw()` докато цикълът не се прекъсне. При някои енджини и framework-ове освен `Update()`, има и отделна функция за обработка на входни данни от клавиатура, мишка, джойстик или други, но в *XNA* това се обработва в `Update()`. Накрая в `Dispose()` се освобождават всички заети ресурси. В `Update()` се правят всички изчисления, изкуствен интелект, засичане на колизии, физика и други. В `Draw()` се извършва само рисуването. Идеята е да се отдели това, което потребителят не вижда – логиката, от това което вижда – изрисуването на моделите и картинките на екрана.

XNA предоставя основни базови класове, които съдържат тези методи. Тези абстрактни класове се наследяват от програмистите и се допълват с необходимата функционалност за съответната игра.

2.3.1. Microsoft.Xna.Framework.GameComponent

Този клас няма `Draw()` метод, следователно е подходящ за компоненти от играта, които се обновяват, но не се рисуват. Например изкуствен интелект, физика, камера, клас-контролер и други.

2.3.2. Microsoft.Xna.Framework.DrawableGameComponent

Най-често използваният клас. Съдържа всички от основните методи. Подходящ за компоненти, които се обновяват и рисуват. Например за главният герой, за враговете, за интерфейса на играта и всичко останало, което изпълнява някаква функция, но и се рисува.

2.3.3. Microsoft.Xna.Framework.Game

Това е основният клас на всяка игра и задължително го има. Той притежава всички от гореспоменатите методи. В неговите `Initialize()`, `Update()` и `Draw()` се викат съответните методи на всички останали компоненти на играта. Този клас съдържа колекция от тип `Microsoft.Xna.Framework.GameComponentCollection`, в която се добавят всички компоненти на играта и след това автоматично им вика методите. Съдържа също контейнер от тип `Microsoft.Xna.Framework.GameServiceContainer`, в който се добавят т.нар. `Service`-и на играта. Те могат да бъдат от всякакъв тип и да се викат от който и да е обект от тип `GameComponent` или `DrawableGameComponent` на играта.

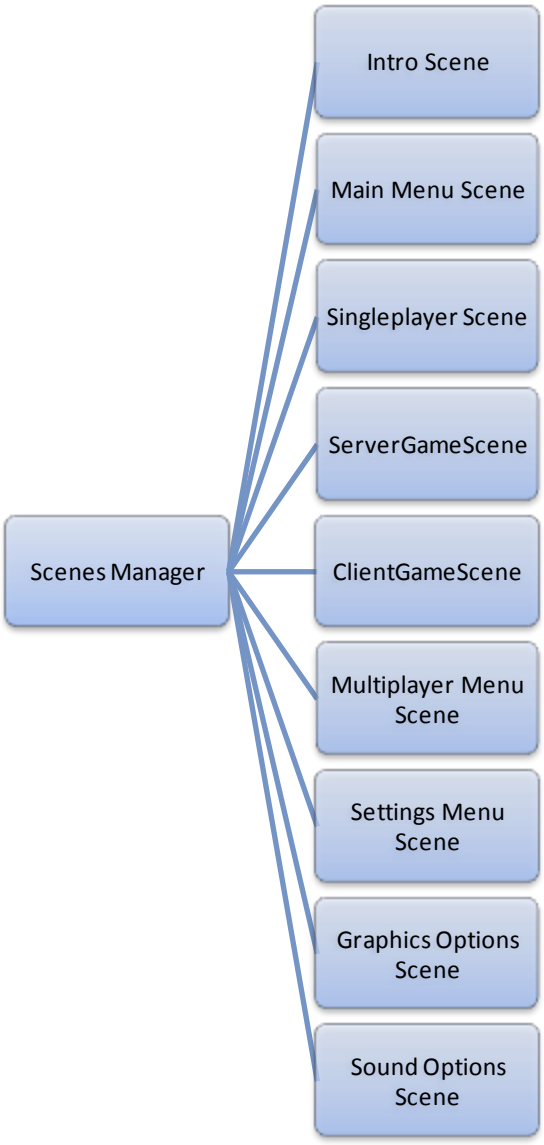
`GameComponent` и `DrawableGameComponent` имат и някои важни *property*-та. `DrawOrder` и `UpdateOrder` съответно определят реда, в който се изрисуват и обновяват. Този ред важи само ако компонентите се намират в колекция от тип `GameComponentCollection`.

Други важни *property*-та са `Enabled` и `Visible`. Те са от булев тип. Стойността на `Enabled` определя дали съответният `Update()` метод на компонента ще се извиква или не. Аналогично `Visible` определя дали `Draw()` метода се вика. Това е полезно при имплементиране на пауза в играта или отваряне на меню за настройки по време на игра.

2.5 Основен алгоритъм на играта.

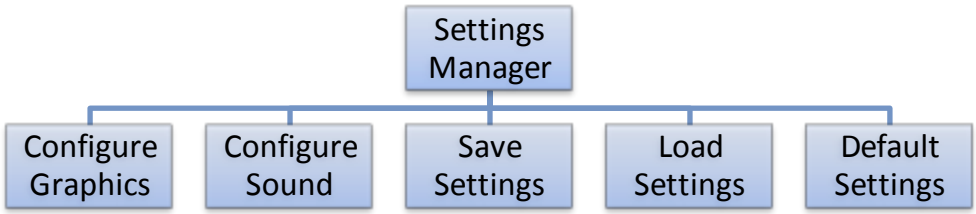
Играта се дели на няколко основни сцени, които съдържат останалите компоненти. Сцените се намират в клас `ScenesManager`, в който винаги има една активна сцена, на която

се викат `Update()` и `Draw()` методите. `ScenesManager` е добавен като `Service` и когато една сцена свърши, или е прекъсната от потребителя, тя казва на `ScenesManager` да пусне друга сцена.



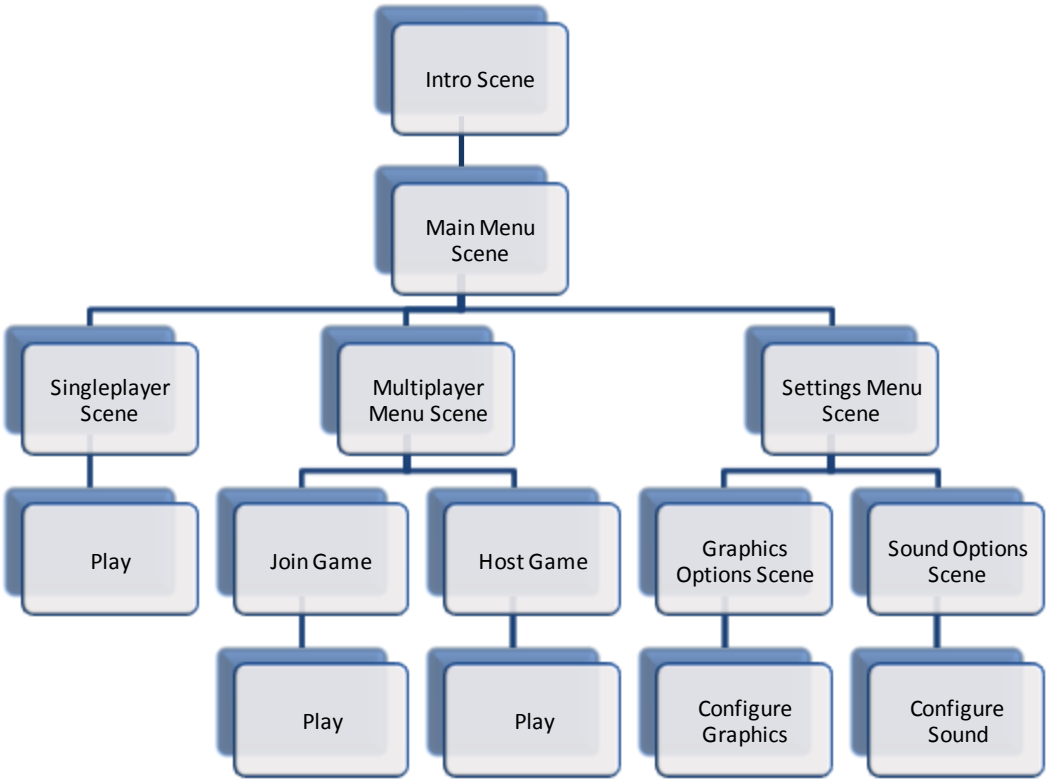
Фиг 2.1. ScenesManager

Конфигурирането на настройките на играта става чрез клас `SettingsManager`, който също е `Service`. След промяната на настройките, той ги записва на файл, за да се заредят отново при следващото стартиране на играта.



Фиг 2.2. SettingsManager

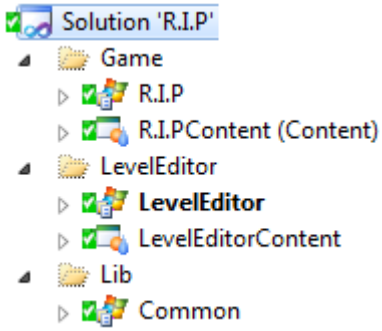
При стартиране на играта се извиква `IntroScene`, която показва името и логото на играта. Като свърши `IntroScene`, тя казва на `ScenesManager` да пусне сцената на главното меню, от където играчът има три възможности – да започне игра в Singleplayer, игра в Multiplayer или да влезе в менюто за настройки на играта, което го препраща към друго меню, където може да избере дали да прави графични или звукови настройки на играта.



Фиг 2.3. Сцените в играта

Глава 3 Реализация на играта и редактора на нива.

Играта и редактора на нива са реализирани в няколко проекта показани на фигурата:



Фиг 3.1. проектите на играта

R.I.P и *R.I.PContent* са съответно проектите на играта, съдържащ специфичните за нея класове и съдържанието на играта – модели, текстури, звуци и шрифтове.

LevelEditor съдържа класовете на редактора на нива. В *LevelEditorContent* са моделите и терена, с които се изграждат нивата на играта.

Common е библиотека, в която има класове общи за играта и редактора на нива.

3.1 Основни и базови класове общи за играта и редактора на нива.

3.1.1 Класът Common.GameBase.Transformation

Целта на този клас е да пази информация за транслациите, ротациите и скалирането на моделите и да изчислява `World` матрицата, необходима за изрисуването на моделите на правилното място. Класът има пет полета, като всички освен `needUpdate` си имат properties:

```
private Vector3 translate;
private Vector3 rotate;
private Vector3 scale;
private Matrix matrix;
private bool needUpdate;
```

`World` матрицата се получава като се умножат скалиращата, ротиращата и транслиращата матрици, получени от съответните вектори.

```
if( needUpdate )
{
	world = Matrix.CreateScale(scale) *
		Matrix.CreateRotationX(MathHelper.ToRadians(rotate.X)) *
		Matrix.CreateRotationY(MathHelper.ToRadians(rotate.Y)) *
		Matrix.CreateRotationZ(MathHelper.ToRadians(rotate.Z)) *
		Matrix.CreateTranslation(translate);
	needUpdate = false;
}

return world;
```

needUpdate показва дали е необходимо матрицата да се смята наново или не.

3.1.2 Класът Common.GameBase.GameModel

От този клас са инстанциите на всички неанимирани модели в играта. Той наследява от Microsoft.Xna.Framework.DrawableGameComponent и съдържа още Model, Transformation, името на файла, от който е зареден модела и reference към камерата на играта, тъй като са необходими Projection и View матриците на камерата за изрисуването на модела. Model представлява самият триизмерен модел, който се зарежда от файл. Класът също има и *property* за BoundingSphere на модела. Тя е нужна за засичане на колизии.

```
		private Model model;
		private Transformation transform;
		private Camera camera;
		private string modelName;


...
		public GameModel(Game game, string filename)
			: base(game)
		{
			modelName = filename;

			model = Game.Content.Load<Model>(GameAssetsPath.MODELS+ filename);
			transform = new Transformation();
			transform.Translate = Vector3.Zero;
			camera = Game.Services.GetService(typeof(Camera)) as Camera;
		}
...
		public BoundingSphere BoundingSphere
		{
			get { return model.Meshes[0].BoundingSphere; }
		}
```

Изрисуването на модела става като се обходят всички Mesh-ове на модела, всички ефекти на Mesh-овете и им се присвоят World матрицата от класа Transformation и View и Projection матриците на камерата. Всички неанимирани модели в играта имат само по един Mesh, затова и *property*-то за BoundingSphere се взима индекс 0.

```
public override void Draw(GameTime gameTime)
{
	base.Draw(gameTime);

	foreach( ModelMesh mesh in model.Meshes )
	{
		foreach( BasicEffect effect in mesh.Effects )
		{
			effect.World = transform.World;
			effect.View = camera.View;
			effect.Projection = camera.Projection;
			effect.EnableDefaultLighting();
		}
		mesh.Draw();
	}
}
```

3.1.3 Класът Common.GameBase.LevelImporter

LevelImporter служи за зареждането на нива. В играта той се използва, за да се заредят всички модели в сцената, а в редактора на нива служи за отваряне на създадени нива, които трябва да се редактират. Нивата на играта се описват с *XML* файлове, в които има информация за позицията, ротацията и размера на всеки модел, както и дали този модел трябва да бъде третиран като физичен. Този клас има един публичен метод, който връща списък с обекти от тип GameModel.

```
	public List<GameModel> Import(string file)
	{
	    models = new List<GameModel>();
	    reader = new XMLTextReader(file);

	    while( reader.Read() )
	    {
	        ReadName();
	        ReadIsPhysicsObject();
	        ReadTransformation();
	    }
	    reader.Close();
	    return models;
	}
```

`ReadName()`, `ReadIsPhysicsObject()`, `ReadTransformation()` са `private` методи, които прочитат стойността в *XML* файла, разделят `string`-а и парсват необходимите стойности.

```
private void ReadTransformation()
{
       if( reader.Name == "Translate" )
       {
              reader.Read();
              string [] xyz = reader.Value.Split( new char[] {' '}, 3);
              Vector3 translate = new Vector3(float.Parse(xyz[0]),
                                              float.Parse(xyz[1]),
                                              float.Parse(xyz[2]));
                tempModel.Transformation.Translate = translate;
                reader.Read();
            }
...
```

Аналогично се прочитат и векторите `rotate` и `translate`.

```
        private void ReadIsPhysicsObject()
        {
            if( reader.Name == "IsPhysicsObject" )
            {
                reader.Read();
                tempModel.IsPhysicsModel = bool.Parse(reader.Value);
                reader.Read();
            }
        }

        private void ReadName()
        {
            if( reader.Name == "Name" )
            {
                reader.Read();
                tempModel = new GameModel(game, reader.Value);
                reader.Read();
            }
        }
```

3.1.4 Класът Common.Misc.DefaultSettings

`DefaultSettings` е класът, в който се намират настройките по подразбиране на играта. Те се зареждат, когато от играчът влезе в менюто за настройка на звука или графиката и избере *Default*.

3.1.5 Класът Common.Misc.GameAssetsPath

В този клас са записани директориите, от които се прочитат всички ресурси на играта – модели, текстури, шрифтове и звуци.

```
namespace Common.Misc
{
	public static class GameAssetsPath
	{
		public static readonly string EFFECTS  = "Effects/";
		public static readonly string FONTS    = "Fonts/";
		public static readonly string MODELS   = "Models/";
		public static readonly string TEXTURES = "Textures/";
		public static readonly string SOUNDS   = "Sounds/";
		public static readonly string IMAGES   = "Images/";
		public static readonly string PLAYERS  = "Models/Players/";
		public static readonly string ENEMIES  = "Models/Enemies/";
	}
}
```

3.1.6 Класът Common.Misc.MenuChoices

Аналогично на `GameAssetsPath`, но държи имената на менютата. Това е полезно, когато проверяваме върху кое меню е курсорът на мишката – не е нужно да пишем ръчно името на менюто, да използваме този клас.

3.1.7 Класът Common.Misc.ScenesNames

Аналогично на предишните два класа, но в случая са записани имената на сцените.

3.2 Класове в редактора на нива.

3.2.1 Класът LevelEditor.Misc.LevelExporter

Този клас записва вече създадени нива в *XML* файл. За целта в единствения си публичен метод получава като аргумент името на файла, който да запише и списък с модели.

```
	public void Export(string file, List<GameModel> models)
	{
	    if( file.EndsWith(".XML") )
	        writer = new XMLTextWriter(file, null);
	    else
	        writer = new XMLTextWriter(file + ".XML", null);
```

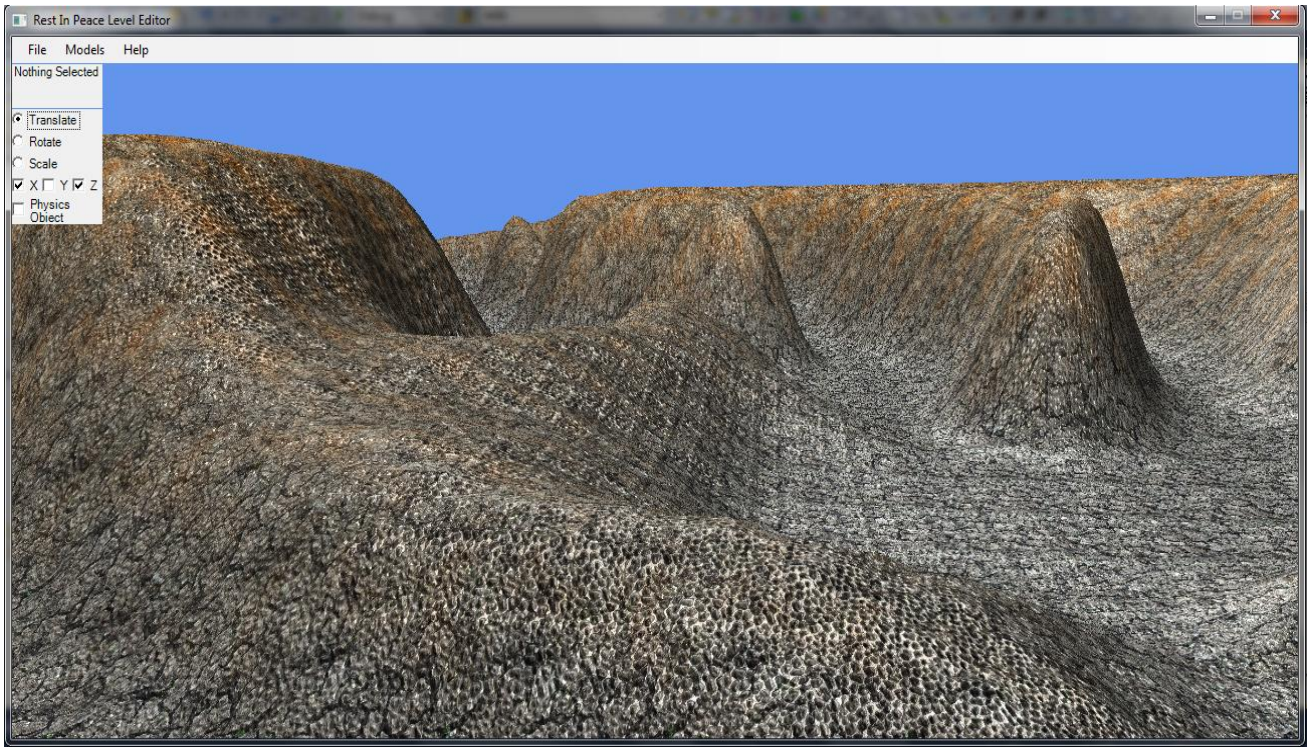
```
            writer.WriteStartDocument();

            writer.WriteStartElement("Level");
            foreach( GameModel model in models )
            {
                WriteModel(model);
            }
            writer.WriteEndElement();

            writer.WriteEndDocument();
            writer.Close();
        }
```

Записването във файл става чрез класът `System.XML.XMLTextWriter`. Задължително в началото на записването и в края трябва да се изпозват съответно методите `WriteStartDocument()` и `WriteEndDocument()`. Всички елеметни в *XML* документа се записват с методите `WriteStartElement()`, `WriteValue()` и `WriteEndElement()`.

```
private void WriteName(GameModel model)
{
   writer.WriteStartElement("Name");
   writer.WriteValue(model.Name);
   writer.WriteEndElement();
}
```

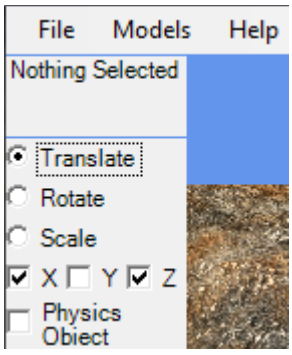
3.2.2 Класът LevelEditor.Misc.LevelEditorForms

`LevelEditorForms` добавя към стандартния XNA прозорец за игри менюта, бутони и други форми. Този клас е тясно обвързан с класа `ModelsManipulator`, който реално прави модификации на моделите.



Фиг. 3.1. Прозорец на редактора на нива

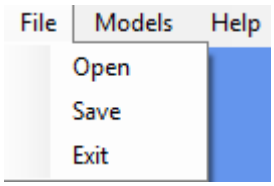
В горния ляв ъгъл на прозореца се намират менютата и бутоните, с които се определя как да бъде редактиран или къде да бъде поставен на терена.



Фиг. 3.2. Менюта и бутони на редактора на нива

3.2.2.1 Менюто File

В менюто File има три възможности *Open*, *Save* и *Exit*.



Фиг. 3.3. Менюто File

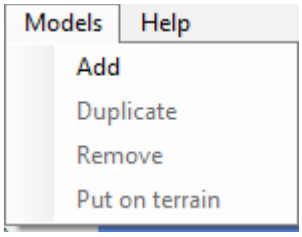
Open и *Save* отварят диалогови прозорци съответно за запазване или зареждане на ниво. За целта се използват съответно `LevelExporter` и `LevelExporter`. При *Open* се прочита файла с нивото, изчистват се всички заредени модели, обхождат се всички модели и се добавят в контейнера на модели.

```
private void open_Click(object o, EventArgs e)
{
   OpenFileDialog openDialog = new OpenFileDialog();
   openDialog.Title = "Open Level";
   openDialog.Filter = "XML Files *.XML|";

   if( openDialog.ShowDialog() == DialogResult.OK )
   {
      modelsManip.Models.Clear();
      List<GameModel> models = levelImporter.Import(openDialog.FileName);
      foreach( GameModel model in models )
      {
          modelsManip.Models.Add(model);
      }
   }
}
```

При избиране на Save на `LevelExporter` се подава името на файла от файловия диалог и контейнера с модели.

```
private void save_Click(object o, EventArgs e)
{
    SaveFileDialog saveDialog = new SaveFileDialog();
    saveDialog.Title = "Save Level";
    saveDialog.Filter = "XML Files *.XML|";
    if( saveDialog.ShowDialog() == DialogResult.OK )
    {
       levelExporter.Export(saveDialog.FileName, modelsManip.Models);
    }
}
```

3.2.2.2 Менюто Models



Фиг. 3.4. Менюто Models

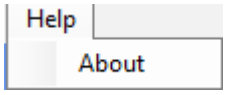
Add добавя нов модел в сцената, като аналогично на отварянето на карта се отваря диалогов прозорец, с който се избира файла, който искаме да заредим. Останалите отговарят съответно за копирането на модел (вместо да го зареждаме отново), изтриването и поставянето на модел върху терена. Тези менюта са неактивни, ако в сцената няма никакви модели или нито един не е селектиран. Това се проверява при всяко извикване на `Update()` метода.

```
...
if( modelsManip.Models.Count <= 0 || modelsManip.SelectedModel == null )
{
	duplicateModel.Enabled = false;
	removeModel.Enabled = false;
	putOnTerrain.Enabled = false;
}
else
{
	duplicateModel.Enabled = true;
	removeModel.Enabled = true;
	putOnTerrain.Enabled = true;
}
...
```

Поставянето на модел върху терена става чрез взимане на височината на терена в точката, където се намира модът и се транслира по Y с необходимата стойност.

```
public void PutOnTerrain()
{
   Terrain terrain = Game.Services.GetService(typeof(Terrain)) as Terrain;
   Vector3 modelPos = SelectedModel.Transformation.Translate;

   modelPos.Y = terrain.GetHeight(modelPos);
   models[selectedModelIndex].Transformation.Translate = modelPos;
}
```

3.2.2.3 Менюто Models



Фиг. 3.5. Менюто Help

About дава информация за версията на редактора на нива.

3.2.3 Класът LevelEditor.Misc.ModelsManipulator

`ModelsManipulator` отговаря за подреждането и редактирането на моделите в нивото. Има три основни действия – *Translate*, *Rotate* и *Scale*, както и *CheckBox* бутони, определящи по кои оси (*X*, *Y* и/или *Z*) да се извършват съответните промени. Чрез *Scale* се контролира размера на моделите, *Rotate* е за тяхната ориентация в *3D* пространството, а *Translate* определя позицията им. Камерата може да се движи напълно свободно в пространството.

Моделите се селектират чрез левия бутон на мишката. Проверката дали при натискане на бутона курсорът на мишката е върху модел става чрез пускане на лъч в *3D* пространството от позицията на мишката, след което се обхождат всички модели и се проверява се дали този лъч пресича `BoundingBox`-а на някой от тях. Ако да, избраният модел си затъмнява цвета за да покаже, че е селектиран и неговият индекс от списъка с модели се запазва в `selectedModelIndex`.

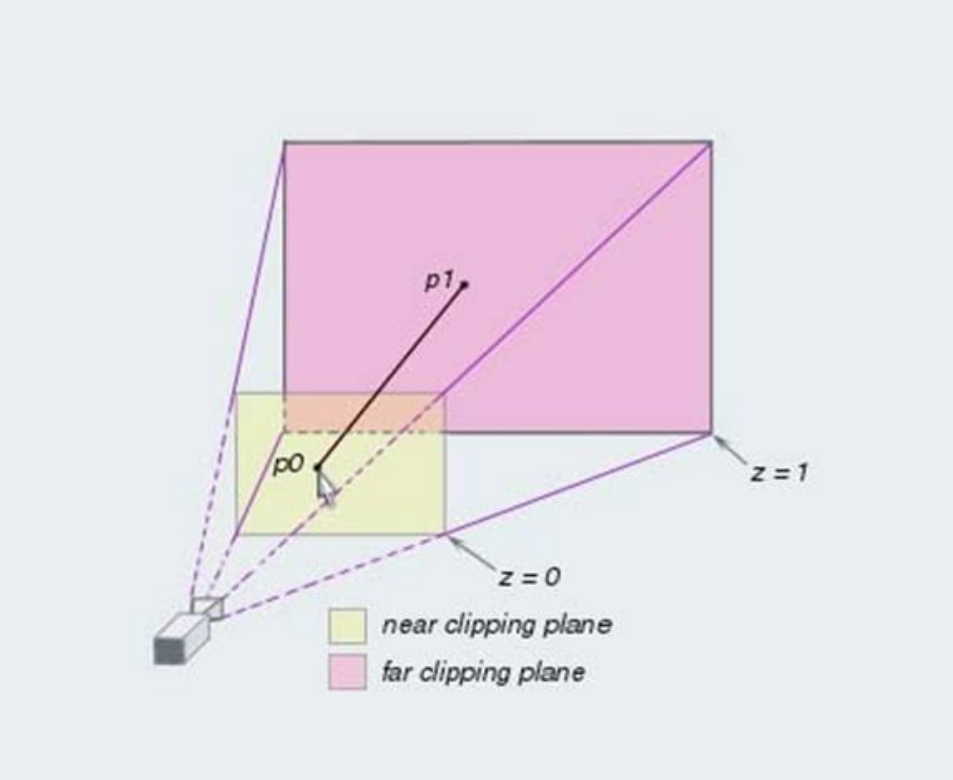
```
if( Mouse.GetState().LeftButton == ButtonState.Pressed &&
      prevMouseState.LeftButton == ButtonState.Released )
{
      PickAnObject();
}
```

В метода `PickAnObject()` се извършват процесите описани по-горе. В него се викат няколко метода – `CalculateRayFromScreen()`, който пуска лъч от точка на екрана и `IntersectDistance()`, който връща разстоянието от лъча до избрания модел, ако има такъв, ако няма връща `null`.

```
private void CalculateRayFromScreen(Vector2 vec, out Vector3 nearPoint, out Ray ray,
                                    out Vector3 direction)
```

```
{
      Viewport viewport = Game.GraphicsDevice.Viewport;

      nearPoint = viewport.Unproject(new Vector3(vec.X, vec.Y, 0),
                                          camera.Projection,
                                          camera.View,
                                          Matrix.Identity);

      Vector3 farPoint = viewport.Unproject(new Vector3(vec.X, vec.Y, 1),
                                          camera.Projection,
                                          camera.View,
                                          Matrix.Identity);

      direction = farPoint - nearPoint;
      direction.Normalize();

      ray = new Ray(nearPoint, direction);
}
```

За да се постигне селектирането, освен позицията на мишката ни трябват и `Projection` и `View` матрицата на камерата и `Viewport`-а. Това е така, защото за да разберем къде е лъчът в сцената, трябва да знаем как е нагласена сцената. Тя се определя от тези три параметъра.



Фиг. 3.6. Near and far clipping plane

Следващото нещо, което правим е да създадем точка в близкия *clipping plane* и точка в далечния *clipping plane*. Това става чрез `Viewport.Unproject()` метода. Тези две точки са разположени на лъча, който искаме да изчислим. С други думи двете точки са точно под курсора на мишката, но едната е много близко до екрана, а другата – много далече. Посоката на лъча се получава като извадим двете точки, а самият лъч от точката в близкия clipping plane и нормализирания вектор на посоката.

Другият метод - `IntersectDistance(BoundingSphere sphere)` проверява дали пуснатият лъч пресича `BoundingSphere`-а на модел. Този метод се използва при обхождане на всички модели – на всеки модел `BoundingSphere`-а се подава като аргумент и ако върне стойност различна от `null`, значи мишката се намира върху модела.

```
        private float? IntersectDistance(BoundingSphere sphere)
        {
                    Ray ray = CalculateRayFromScreen(currentMousePos2D);
```

```
            return ray.Intersects(sphere);;
        }
```

`float?` означава, че на променливата от този тип може да бъде присвоявана стойност `null`.

След като сме селектирали модел, от радио бутоните *Translate*, *Rotate*, *Scale* може да изберем коя от тези трансформации да правим, от *CheckBox*-овете *X*, *Y* и *Z* да определим по кои оси да стават промените, а от *CheckBox*-а *PhysicsModel* запаметяваме дали селектираният модел да бъде третиран като физичен в играта или не. В Update() метода на `ModelsManipulator` се отчита `deltaMouse` – поле, което представлява двуизмерен вектор в който се записва промяната в позицията на мишката по X и Y, както и текущата позиция на курсора на екрана в полето `currentMousePos2D`. Тези променливи са ни нужни при трансформациите на моделите.

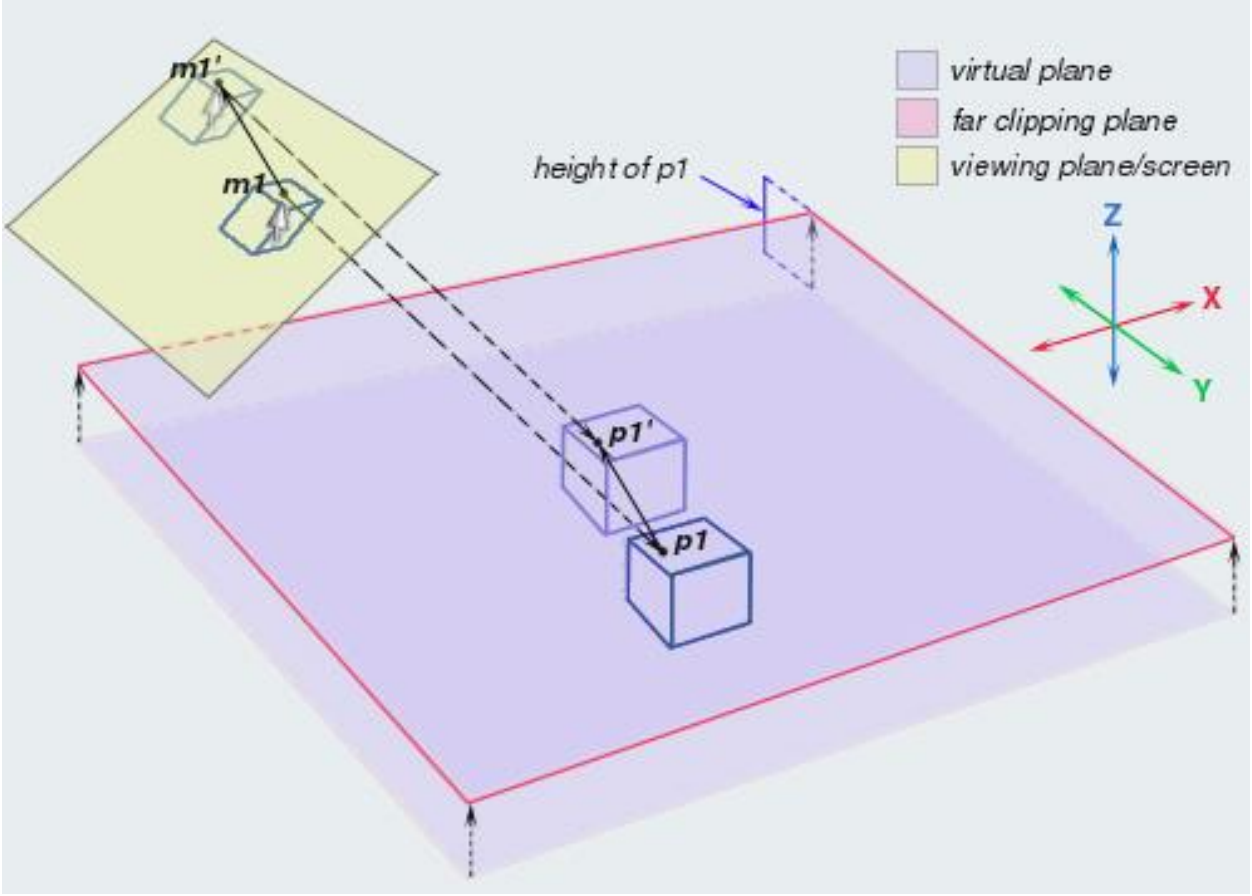
```
private void UpdateMouseVars()
{
       currentMousePos2D = new Vector2(currentMouseState.X, currentMouseState.Y);
       deltaMouse.X = currentMouseState.X - prevMouseState.X;
       deltaMouse.Y = currentMouseState.Y - prevMouseState.Y;
}
```

Трите вида трансформации се извършват в три отделни метода `HandleTranslate()`, `HandleRotate()` и `HandleScale()`, които се викат в зависимост от избрания радио бутон. В `HandleRotate()` и `HandleScale()` съответно на `Transformation.Rotate` или `Transformation.Scale` вектора на избрания модел се добавя ротацията или скалирането по избраната(ите) ос(и).

```
....
        private void HandleRotate()
        {
            Vector3 rotation = new Vector3();
            if( axes.X )
                rotation.X = deltaMouse.Y * mouseSensitivity;
            if( axes.Y )
                rotation.Y = deltaMouse.X * mouseSensitivity;
            if( axes.Z )
                rotation.Z = deltaMouse.X * mouseSensitivity;

            models[selectedModelIndex].Transformation.Rotate += rotation;
        }

....
```

```
        private void HandleScale()
        {
            if( axes.X && axes.Y && axes.Z )
            {
                models[selectedModelIndex].Transformation.Scale +=
                                        new Vector3(deltaMouse.X * mouseSensitivity);
                return;
            }

            Vector3 scale = new Vector3();
            if( axes.X )
                scale.X = deltaMouse.X * mouseSensitivity;
            if( axes.Y )
            {
                if( axes.Z )
                    scale.Z = deltaMouse.X * mouseSensitivity;
                scale.Y = -deltaMouse.Y * mouseSensitivity;
            }
            if( axes.Z && !axes.Y )
                scale.Z = deltaMouse.Y * mouseSensitivity;

            models[selectedModelIndex].Transformation.Scale += scale;
        }
...
```

`HandleTranslate()` е малко по-сложна функция, тъй като искаме да движим моделите заедно с движението на мишката. Целият алгоритъм е показан на фиг 3.7:



Фиг. 3.7. Алгоритъм за транслация на модели

Описание на алгоритъма:

1. Пускаме лъч от *m1* (позицията на мишката на екрана при натискане на левия бутон) и изчисляваме *p1* (пресечната точка на лъча, пуснат от мишката и `BoundingSphere`-а на селектирания модел). Това изчисление става в `PickAnObject()` метода и резултата го записваме в `initialMousePos3D`. Позицията на мишката я имаме по всяко време, защото тя се обновява на всеки кадър в метода `UpdateMouseVars()`.

```
initialMousePos3D = nearPoint + direction * (float)distance;
virtualPlane = CreateRequiredPlane(initialMousePos3D);
```

2. Създаваме виртуалната равнина, на височината пресечната точка на лъча с модела (*p1*), която лежи на равнината. На фиг 3.7. е показано в случая, когато икаме да

движим модела по *X* и *Y*. Това се определя от това кои *CheckBox*-ове за *X*, *Y* и *Z* сме избрали.

```
private Plane CreateXZPlane(Vector3 pos)
{
    return new Plane(
                pos,
                new Vector3(pos.X * 2, pos.Y, pos.Z),
                new Vector3(pos.X, pos.Y, pos.Z * 2));
}

private Plane CreateYZPlane(Vector3 pos)
{
    return new Plane(
                pos,
                new Vector3(pos.X, pos.Y * 2, pos.Z),
                new Vector3(pos.X, pos.Y, pos.Z * 2));
}

private Plane CreateXYPlane(Vector3 pos)
{
    return new Plane(
                pos,
                new Vector3(pos.X, pos.Y * 2, pos.Z),
                new Vector3(pos.X * 2, pos.Y, pos.Z));
}

private Plane CreateRequiredPlane(Vector3 pos)
{
    if( axes.X && axes.Z )
        return CreateXZPlane(pos);
    if( Axes.X && Axes.Y )
        return CreateXYPlane(pos);
    if( axes.Y && axes.Z )
        return CreateYZPlane(pos);

    return CreateXZPlane(pos);
}
```

3. При движение на мишката пускаме лъч от новата позиция на мишкатана екрана до (*m1'*) виртуалната равнина и намираме пресечната точка (*p1'*). Това става в `HandleTranslate()`, като записваме резултата в `currentMousePos3D`.

4. Изчисляваме вектора на движение като извадим от `currentMousePos3D` `initialMousePos3D` и го прибавяме към матрицата за транслации на селектирания модел.
5. Стъпки 3. и 4. се повтарят докато не изберем друг модел или друга трансформация. За целта след стъпка 4. трябва да се подготвим за следващите итерации. Това става като присвоим на `initialMousePos3D` стойността на `currentMousePos3D`.

```
private void HandleTranslate()
{
	if( SelectedModel != null &&
		currentMouseState.LeftButton == ButtonState.Pressed &&
		prevMouseState.LeftButton == ButtonState.Pressed )
	{
		Vector3 currentMousePos3D;
		Ray ray;
		Vector3 nearPoint;
		CalculateRayFromScreen(currentMousePos2D, out nearPoint, out ray,
							out currentMousePos3D);
		float? distance = ray.Intersects(virtualPlane);
		if( distance != null )
		{
			currentMousePos3D = nearPoint + currentMousePos3D * (float)distance;
			Vector3 movementVec = currentMousePos3D - initialMousePos3D;
			models[selectedModelIndex].Transformation.Translate += movementVec;
			initialMousePos3D = currentMousePos3D;
		}
	}
}
```

3.2.4 Класът LevelEditor.LevelEditor

В този клас се създават и инициализират останалите класове от проекта и се добавят като Service-и `LevelEditorForms` и `ModelsManipulator`. В `Update()` се обновяват, а в `Draw()` се обхожда контейнера с модели и се рисуват.

3.2.5 Класът LevelEditor.Program

Това е началната точка на програмата. Създава се инстанция на `LevelEditor` и се стартира.

```
        static void Main(string[] args)
        {
            using (LevelEditor levelEditor = new LevelEditor())
            {
                levelEditor.Run();
            }
        }
```

3.3 Класове в играта.

3.3.1 Класът R.I.P.Managers.ScenesManager

ScenesManager съдържа колекция от различните сцени на играта – интро, меню, настройки, single и multiplayer, индекс на активната сцена и самата активна сцена, методи за тяхното сменяне и *property*-та на полетата.

```
public void SetActiveScene(int index)
{
       activeSceneIndex = index;
       activeScene = scenes[scenes.Keys[activeSceneIndex]];
}

public void SetActiveScene(string id)
{
       SetActiveScene(id, false, true);
}

public void SetActiveScene(string id, bool disposePrev, bool reInitNew)
{
       if( disposePrev )
       {
              if( activeScene != null )
              {
                     scenes.RemoveAt(activeSceneIndex);
              }
       }

       activeSceneIndex = scenes.IndexOfKey(id);
       activeScene = scenes[id];

       if( reInitNew )
              activeScene.Initialize();
}
```

Активната сцена може да се сменя по три разлини начина – чрез индекс, чрез името на сцената и чрез името на сцената и възможност за реинициализация на новата активна сцена. Последното ни трябва в някои особени случаи. Например: играчът е започнал игра в *Single Player*, но излиза в главното меню. Тогава има няколко възможности – ако избере *Resume* играта продължава от там, от където е била паузирана. В този случаи метода се

вика със стойност `false` на последния аргумент. Но ако избере *Single Player,* трябва да се стартира нова, като се извика отново `Initialize()` метода на сцената.

В метода `Add(string id, AScene scene)` в контейнера за сцените се добавя нова сцена. Ако той е празен, тогава първата добавена става и активна. `Remove(string id)` изтрива избраната сцена, а `Clear()` изчиства всички сцени.

```
public void Add(string id, AScene scene)
{
	if( scenes.ContainsKey(id) )
	{
		scenes.Remove(id);
	}

	scenes.Add(id, scene);

	if( activeScene == null )
	{
		SetActiveScene(id);
	}
}

public void Remove(string id)
{
	scenes.Remove(id);
}

public void Clear()
{
	scenes.Clear();
	activeSceneIndex = -1;
	activeScene = null;
}
```

3.3.2 Класът R.I.P.Managers.SoundManager

`SoundManager` съдържа музиката, всички звуци, променливи за силата на звука, музиката и ефектите и методи и *property*-та за тяхното пускане и спиране. XNA използва Windows Media Player за звуци и музика и класовете `Song` и `SoundEffect` си имат собствени методи за пускане, паузиране и спиране. Чрез този клас само се избира коя песен или звуков ефект да бъде пуснат.

3.3.3 Класът R.I.P.Managers.SettingsManager

`SettingsManager` се грижи за звуковите и графичните настройки на играта. Чрез него може да се контролира силата на зука, резолюцията на екрана, дали играта да е на пълен екран, дали да има вертикална синхронизация и други. За да има контрол върху звука и графичните настройки `SettingsManager` се нуждае от `SoundManager`-а на играта и вграденият в XNA клас `GraphicsDeviceManager`, който се създава още при инициализацията налседения `Game` клас.

```
graphicsManager =
       Game.Services.GetService(typeof(GraphicsDeviceManager)) as GraphicsDeviceManager;
soundManager = Game.Services.GetService(typeof(SoundManager)) as SoundManager;
```

За контролиране на силата на звука се използват *property*-тата на класа `SoundManager`.

```
public void SetSoundSettingsDefault()
{
       soundManager.MusicVolume = DefaultSettings.MUSIC_VOLUME;
       soundManager.EffectsVolume = DefaultSettings.EFFECTS_VOLUME;
       soundManager.VoicesVolume = DefaultSettings.VOICES_VOLUME;
}

public void DecreaseMusicVolume()
{
       if( soundManager.MusicVolume > 0 )
              soundManager.MusicVolume -= changeVolBy;
}

public void DecreaseEffectsVolume()
{
       if( soundManager.EffectsVolume > 0 )
              soundManager.EffectsVolume -= changeVolBy;
}

public void IncreaseMusicVolume()
{
       if( soundManager.MusicVolume < 1 )
              soundManager.MusicVolume += changeVolBy;
}

public void IncreaseEffectsVolume()
{
       if( soundManager.EffectsVolume < 1 )
              soundManager.EffectsVolume += changeVolBy;
}
```

При първото стартиране се прочитат различните резолюции, поддържани от монитора и видеокартата и се записват в списък, за да може когато се променя резолюцията да се избират само измежду поддържани. Това става в метода `GetDisplayModes();`

```
private void GetDisplayModes()
{
       DisplayMode currentDisplayMode =
                              graphicsManager.GraphicsDevice.Adapter.CurrentDisplayMode;
       displayModes = new DisplayMode[CountDisplayModes()];

       int count = 0;
       foreach( DisplayMode displayMode in
                              graphicsManager.GraphicsDevice.Adapter.SupportedDisplayModes )
       {
              if( displayMode.Format == currentDisplayMode.Format &&
                     displayMode.Width >= 800 && displayMode.Height >= 480 )
              {
                     displayModes[count] = displayMode;
                     count++;
              }
       }
}
```

След като са прочетени и записани в масив резолюциите от тях може да бъде избирана активна. Това става чрез промяната на индекса на избраната резолюция. След промяната, обаче трябва да се коригира *Aspect ratio*-то на камерата, като се изчисли отношението между широчината и височината на екрана на играта.

```
public void CycleResolutions()
{
   displayModeIndex++;
   if( displayModeIndex == displayModes.Length )
   {
       displayModeIndex = 0;
   }

       graphicsManager.PreferredBackBufferWidth = displayModes[displayModeIndex].Width;
       graphicsManager.PreferredBackBufferHeight = displayModes[displayModeIndex].Height;
       graphicsManager.ApplyChanges();

       UpdateCamera();
}
```

```
private void UpdateCamera()
{
    ThirdPersonCamera camera = Game.Services.GetService(typeof(ThirdPersonCamera))
                                                                  as ThirdPersonCamera;
    if( camera != null )
    {
        camera.AspectRatio =
          (float)Game.Window.ClientBounds.Width / (float)Game.Window.ClientBounds.Height;
   }
}
```

Другата възможна графична настройка е опцията за включване и изключване на вертикалната синхронизация (*VSync*). Тя решава *Screen Tearing* проблема, като забранява на играта да прави повече кадри в секунда от честотата на опресняване на монитора и също възпира видео картата да прави нещо видимо за графичната памет докато мониторът не завърши текущия цикъл на опресняване.

Фиг. 3.8. Screen tearing.

Тази настройка става в `GraphicsDeviceManager`-а на *XNA*. След каквато и да е настройка трябва да се извика и метода `Apply()`, за да се отразят настройките.

```
public void ToggleVSync()
{
       graphicsManager.SynchronizeWithVerticalRetrace =
                                    !graphicsManager.SynchronizeWithVerticalRetrace;
       graphicsManager.ApplyChanges();
}
```

3.3.4 Класът R.I.P.Scenes.AScene

Това е абстрактен клас, от който наследяват различните сцени на играта. Съдържа две основни функции – `Hide()` и `Show()`, които са нужни за активиране/деактивиране и показване/скриване тоест дали `Draw()` и `Update()` да се викат.

```
public void Show()
{
	Enabled = true;
	Visible = true;
}

public void Hide()
{
	Enabled = false;
	Visible = false;
}
```

3.3.5 Класът R.I.P.Menus.AMenu

`AMenu` също е абстрактен клас. Той държи базовата функционалност и полета на едно меню. Съдържа масив с имената и позициите на отделните елементи или бутони на менюто. За да се постигне ефект също има и два цвята – единияте за изрисуване на елементите на менюто, а другият се използва когато мишката е върху определен елемент. За същата цел менюто бавно си променя цвета при първото стартиране. Тогава той си сменя цвета. В `Update()` метода има проверка дали е избран елемент в менюто и ако да чрез `SoundManager`-а се пуска звук. Класът държи и информация за размера на екрана на играта – с това менютата винаги се позиционират на правилното място на определено разстояние от края на екрана. Съдържа и индекс за това върху кой елемент се намира мишката. Менютата в играта наследяват този клас и изпълват масива с имена и позиции. Класът има и *property*, което дава информация кой елемент е избран.

```
public string Selection
{
	get { return selection; }
}

public int SelectionIndex
{
	get { return selectionIndex; }
}

private void ObtainMousePos()
{
```

```
                mousePos.X = Mouse.GetState().X;
                mousePos.Y = Mouse.GetState().Y;
        }

        private void HandleMouseMovement()
        {
                for( int i = 0; i < positions.Length; i++ )
                {
                        if( positions[i].Intersects(mousePos) )
                        {
                                selectionIndex = i;
                                selection = menus[i];
                                return;
                        }
                }
                selectionIndex = -1;
                selection = null;
        }
```

За да се позиционират менютата правилно едно под друго е необходимо да се изчисли размера на най-дългия елемент от менюто. Спримо него елементите се позиционират едно под друго.

```
protected int CalcLongestString()
{
        int longest = 0;
        foreach( string str in menus )
        {
                if( font.MeasureString(str).X > longest )
                        longest = (int)font.MeasureString(str).X;
        }

        return longest;
}

protected int measureStrX(string str)
{
        return (int)font.MeasureString(str).X;
}
```

3.3.6 Класът R.I.P.Scenes.MenuScenes.AMenuScene

Този клас комбинира предишните два абстрактни класа и образува базовата функционалност на една меню сцена в играта. Той наследява от `AScene`, но и съдържа `AMenu`. В него се следи дали бутон на мишката е кликнат веднъж за избиране на елемент от менюто. Като поле той има и `ScenesManager`, който е необходим за всяка меню сцена и чрез който се сменят сцените. Напимер при избиране на елемент от менюто да се отиде в друго меню или при кликване на бутона *Back* да се върне в предишното. За натиснат бутон

в случая се приема ако в предишното състояние бутонът не е бил натиснат, а в текущото е. Това е за да се избегне нежеланото кликване върху бутон от менюто, след бързото преминаване от едно меню в друго.

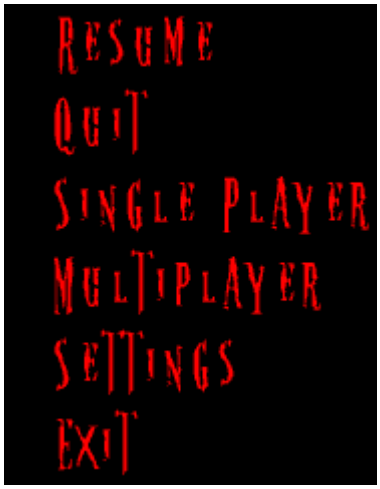
```
protected abstract void HandleSelection();

private void HandleInput()
{
	keysCurrent = Keyboard.GetState();
	HandleSelection();
}

protected bool IsLMBDown()
{
	return mouseCurrent.LeftButton == ButtonState.Pressed &&
		mousePrev.LeftButton != ButtonState.Pressed;
}

protected bool IsRMBDown()
{
	return mouseCurrent.RightButton == ButtonState.Pressed &&
		mousePrev.RightButton != ButtonState.Pressed;
}

public override void Update(GameTime gameTime)
{
	mouseCurrent = Mouse.GetState();
	HandleInput();
	mousePrev = mouseCurrent;

	menu.Update(gameTime);

	base.Update(gameTime);
}
```

3.3.7 Класът R.I.P.Menus.MainMenu

`MainMenu` наследява от `AMenu` и изпълва масива с позициите и имената на менютата. Това става в `override`-натият метод `CalculatePositions()`.

```
protected override void CalculatePositions()
{
	menus = new string[4];
	positions = new Rectangle[4];

	menus[0] = MenuChoices.SINGLE_PLAYER;
	menus[1] = MenuChoices.MULTIPLAYER;
	menus[2] = MenuChoices.SETTINGS;
	menus[3] = MenuChoices.EXIT;

	int longest = CalcLongestString();
	int length = positions.Length;
	for( int i = 0; i < length; i++ )
```

```
        {
                positions[i] = new Rectangle(screenWidth - longest - distanceFromScrX,
                        screenHeight - distBetwChoices * ( length - i ) - distanceFromScrY,
                        measureStrX(menus[i]), measureStrY(menus[i]) );
        }
}
```

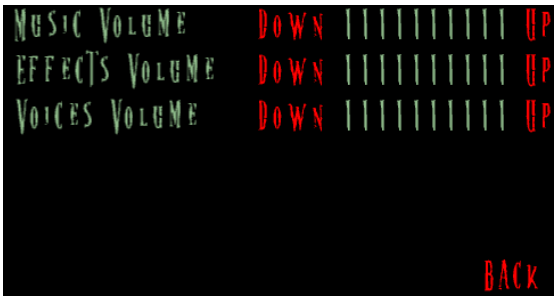


Фиг. 3.9. Main Menu **Фиг. 3.10. Extended Main Menu**

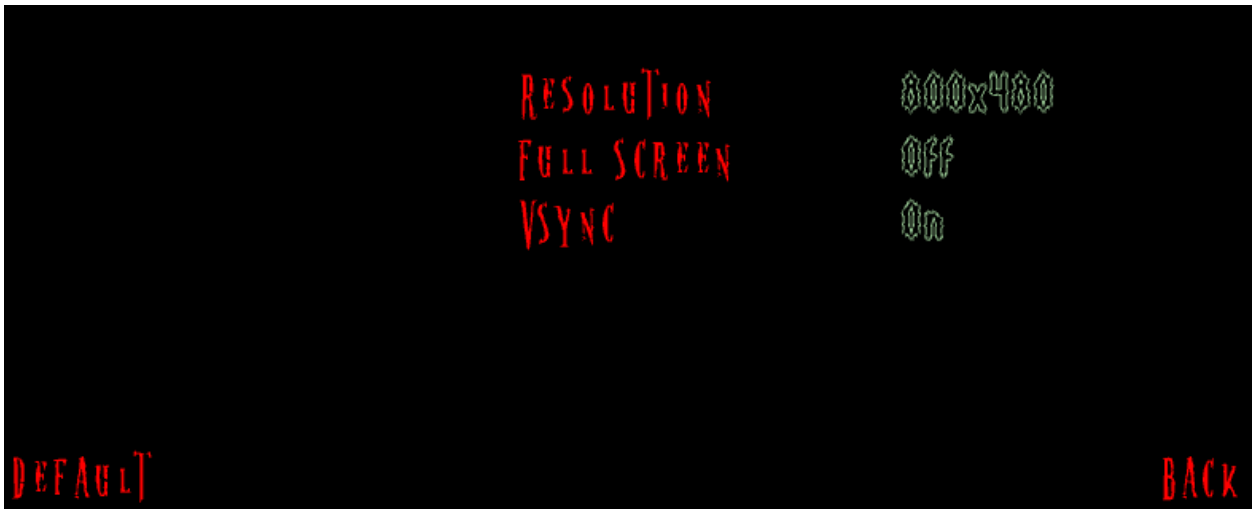
`MainMenuEx` е разширение на `MainMenu`. В него има още два възможни избора – *Resume* и *Quit*. Те се показват само ако има включена игра. Ако изберем Resume, продължаваме играта, с *Quit* прекратяваме играната до момента, а при избиране на *Single Player* или *Multiplayer* се започва нова игра. В менюто *Settings* може да правим настройки и след това пак да продължим играта с *Resume*.

Останалите менюта – `SettingsMenu`, `GraphicsSettingsMenu` и `SoundSettingsMenu` се получават по аналогичен начин.

За функционалността на менютата се грижат съответните меню-сцени.



Фиг. 3.11. Settings Menu **Фиг. 3.12. Sound Settings Menu**



Фиг. 3.13. Graphics Settings Menu

3.3.8 Класът R.I.P.Scenes.MenuScenes.MainMenuScene

Тази сцена се грижи за избора на потребителя в главното меню. Има поле от тип `AMenu` и пуска различните сцени в зависимост от избора на потребителя. Това става като се следи мишката върху кой бутон на менюто е и дали левият бутон е натиснат. Ако има съществуваща сцена от тип `MultiplayerScene` и играчът избере *Single Player*, то multiplayer сцената се изтрива и на нейно място се създава сцена от тип `SingleplayerScene`. Аналогично става когато има съществуваща *Single Player* сцена и се избере *Multiplayer*.

```
private void HandleSelectionDefault(string selection)
{
	if( selection == MenuChoices.SINGLE_PLAYER )
	{
		scenesManager.Add(ScenesNames.SINGLE_PLAYER, new SingleplayerScene(Game));
		scenesManager.Remove(ScenesNames.MULTIPLAYER);
		scenesManager.SetActiveScene(ScenesNames.SINGLE_PLAYER);
	}
	else if( selection == MenuChoices.MULTIPLAYER )
	{
```

```
                scenesManager.Add(ScenesNames.MULTIPLAYER, new MultiplayerScene(Game));
                scenesManager.Remove(ScenesNames.SINGLE_PLAYER);
                scenesManager.SetActiveScene(ScenesNames.MULTIPLAYER);
        }
        else if( selection == MenuChoices.SETTINGS )
                scenesManager.SetActiveScene(ScenesNames.SETTINGS);
        else if( selection == MenuChoices.EXIT )
                Game.Exit();
}
```

Ако в ScenesManager има сцена от тип `MultiplayerScene` или `SingleplayerScene` е менюто на тази сцена е от тип `MainMenuEx`, за да позволява *Resume* и *Quit* опциите.

```
private void HandleSelectionExtended(string selection)
{
        if( selection == MenuChoices.RESUME )
        {
                soundManager.Stop();
                if( scenesManager.Contains(ScenesNames.SINGLE_PLAYER) )
                        scenesManager.SetActiveScene(ScenesNames.SINGLE_PLAYER, false,
                                                                                false);
                else if( scenesManager.Contains(ScenesNames.MULTIPLAYER) )
                        scenesManager.SetActiveScene(ScenesNames.MULTIPLAYER, false, false);
        }
        if( selection == MenuChoices.QUIT )
        {
                if( scenesManager.Contains(ScenesNames.SINGLE_PLAYER) )
                        scenesManager.Remove(ScenesNames.SINGLE_PLAYER);
                else if( scenesManager.Contains(ScenesNames.MULTIPLAYER) )
                        scenesManager.Remove(ScenesNames.MULTIPLAYER);
                        menu = new MainMenu(Game);
        }
}

protected override void HandleSelection()
{
        string selection = menu.Selection;

        if( selection != null )
        {
                if( IsLMBDown() )
                {
                        HandleSelectionDefault(selection);
                        if( menu is MainMenuEx )
                                HandleSelectionExtended(selection);
                }
        }
}
```

От тази сцена може да се отиде в `SingleplayerScene`, `MultiplayerScene` или `SettingsScene`, в зависимост от избора на потребителя или да се излезе от играта при натискане на бутона *Exit*.

3.3.9 Класът R.I.P.Scenes.MenuScenes.SettingsScene

Тази сцена работи по аналогичен начин на MainMenuScene, но с тази разлика, че препраща в `GraphicsSettingsScene` и `SoundSettingsScene`, където могат да се правят настройки.

3.3.10 Класът R.I.P.Scenes.MenuScenes.SoundSettingsScene

Тук могат да се правят настройки за силата на звука и музиката, което става с няколко бутона. Има бутон за настройки по подразбиране и за връщане в предишното меню. За промяна, запазване и връщане на настройките се използват методите от *Service*-а `SettingsManager`.

```
private void HandleChoice(string selection)
{
	if( selection == MenuChoices.DEFAULT )
	{
		settingsManager.SetSoundSettingsDefault();
	}
	if( selection == MenuChoices.BACK )
		scenesManager.SetActiveScene(ScenesNames.SETTINGS);
	if( selection == MenuChoices.DOWN )
	{
		if( menu.SelectionIndex == (int) UpDownButtons.DownMusic )
			settingsManager.DecreaseMusicVolume();
		else if( menu.SelectionIndex == (int) UpDownButtons.DownEffects )
			settingsManager.DecreaseEffectsVolume();
		else if( menu.SelectionIndex == (int) UpDownButtons.DownVoices )
			settingsManager.DecreaseVoicesVolume();
	}
	if( selection == MenuChoices.UP )
	{
		if( menu.SelectionIndex == (int) UpDownButtons.UpMusic )
			settingsManager.IncreaseMusicVolume();
		else if( menu.SelectionIndex == (int) UpDownButtons.UpEffects )
			settingsManager.IncreaseEffectsVolume();
		else if( menu.SelectionIndex == (int) UpDownButtons.UpVoices)
			settingsManager.IncreaseVoicesVolume();
	}
}
```

3.3.11 Класът R.I.P.Scenes.MenuScenes.GraphicsSettingsScene

Този клас отговаря за промяната на графичните настройки от потребителя. От тук може да се сменя резолюцията, да се включва и изключва вертикланата синхронизация и да се превключва между игра на пълен екран или в режим на прозорец.

```
if( IsLMBDown() || IsRMBDown() )
{
	if( menu.Selection == MenuChoices.BACK )
		scenesManager.SetActiveScene(ScenesNames.SETTINGS);
	if( menu.Selection == MenuChoices.FULL_SCREEN )
		settingsManager.ToggleFullScreen();
	if( menu.Selection == MenuChoices.VSYNC )
		settingsManager.ToggleVSync();
	if( menu.Selection == MenuChoices.RESOULUTION )
	{
		if( IsLMBDown() )
			settingsManager.CycleResolutions();
		else if( IsRMBDown() )
			settingsManager.CycleResolutionsBackwards();
		Initialize();
	}
	if( menu.Selection == MenuChoices.DEFAULT )
	{
		settingsManager.SetGraphicSettingsDefault();
		Initialize();
	}
}
```

При смяната на резолюцията отново се вика `Initialize()` метода на самата сцена. Причината за това е, че след смяната на резолюцията бутоните в менюто трябва отново да изчислят позициите си и да се позиционират на правилното място на екрана.

3.3.12 Класът R.I.P.Network.ClientHandler

В този клас има сокет и информация за всеки клиент. Чрез него се праща и получава необходимата информация за и от клиента. За всеки клиент се създава отделен ClientHandler. Всички те се държат в списък който е на сървъра и се обхождат постоянно по време на играта в отделна нишка.

За героите и враговете, които се получават от сървъра има съответно два класа – NetworkPlayer и NetworkEnemy. В тях не се правят никакви изчисления, а само им се писвоява необходимата стойност, получена от сървъра.

3.3.13 Класът R.I.P.Scenes.GameScene.SingleplayerScene

Тук се зареждат терена, камерата, светлината, героя и враговете. Това е сцената в която се играе и се извършва цялата логика на играта в *Singleplayer*.

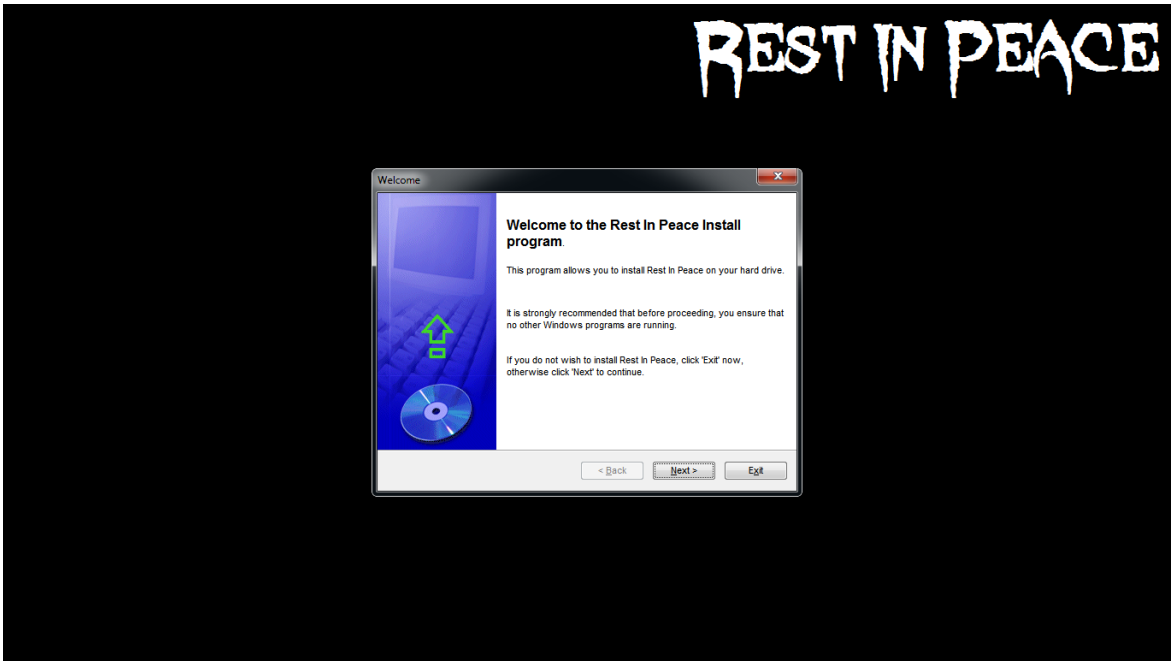
За *Multiplayer* има две други сцени. *ServerGameScene* и *ClientGameScene*. *Multiplayer* режима на играта е тип *Server/Client* а не *Peer-To-Peer*. На сървъра се извършват изчисленията, клиента ги получава и само праща позицията и състоянието си.

Глава 4 Ръководство за потребителя

4.1 Инсталация на продукта в стъпки

1. Продуктът се инсталира като се стартира файлът *Setup.exe*.

2. След това се появява началното меню, което описва каква е целта на инсталатора (Фиг. 4.1.). За да се продължи напред с инсталацията се натиска бутонът *Next*.



Фиг. 4.1. Начален прозорец на инсталацията

3. Следващият прозорец уведомява потребителя, че след инсталацията на самата игра, той ще бъде запитан дали желае да бъдат и инсталирани *Microsoft .NET 4 Client Profile* и *XNA 4 Redistributable*.

ВАЖНО! Ако нямате инсталирани *Microsof.NET 4* и *XNA 4 Redistributable*, е важно те да бъдат инсталирани! В противен случай играта няма да се стартира!

За продължаване отново е нужно да се натисне бутонът *Next*.

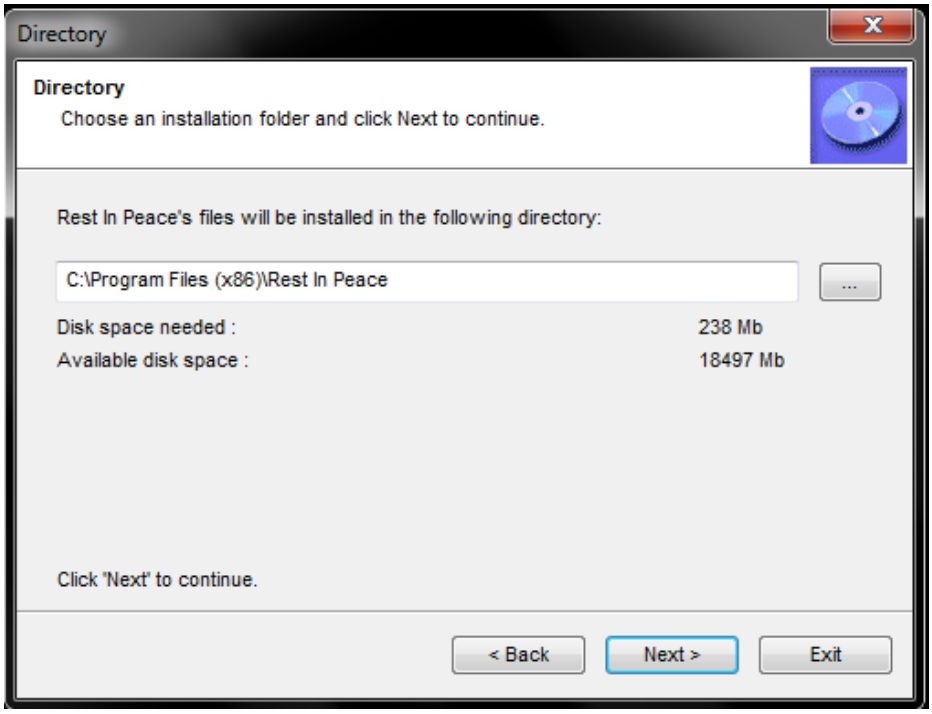
4. Третият прозорец представя копие от GNU GENERAL PUBLIC LICENSE, под който се разпространява играта. Ако потребителят не е съгласен с условията, включени в лиценза –

инсталацията не може да продължи. Ако е съгласен – трябва да избере “I agree with the above terms and conditions” и да натисне бутона *Next*.

5. Следващият прозорец иска информация от потребителя къде да бъде инсталирана играта. Също така показва информация за свободното дисково пространство и дали то е достатъчно, за да бъде инсталирана програмата (Фиг. 4.2). Потребителят може да избере папка от бутона „...“. За да продължи инсталацията се натиска бутонът *Next*. Ако папката не съществува, инсталаторът прави запитване до клиента, дали той желае такава папка да бъде създадена. Ако не - потребителят наново трябва да избере папка.

6. Прозорец, опсиващ продукта за инсталация и мястото, където ще бъде инсталирано. За продължение – бутонът *Start*.

7. След това започва инсталацията на играта. Когато тя приключи се отваря прозорец за инсталация на *Microsoft .NET 4*. Ако потребителят не желае да му бъде инсталиран – може да натисне бутонът *Cancel* и да прекрати инсталацията му. И в двата случая след това се появява нов прозорец за инсталация на *XNA 4 Redistributable*. Инсталацията отново може да бъде прекратена от бутона *Cancel*.



Фиг. 4.2. Инсталация

8. След това започва инсталацията на играта. Когато тя приключи се отваря прозорец за инсталация на *Microsoft .NET 4*. Ако потребителят не желае да му бъде инсталиран – може да натисне бутонът *Cancel* и да прекрати инсталацията му. И в двата случая след това се появява нов прозорец за инсталация на *XNA 4 Redistributable*. Инсталацията отново може да бъде прекратена от бутона *Cancel*.

4.2 Стартиране на играта

Играта се стартира от иконката, на *Desktop - Rest In Peace*.

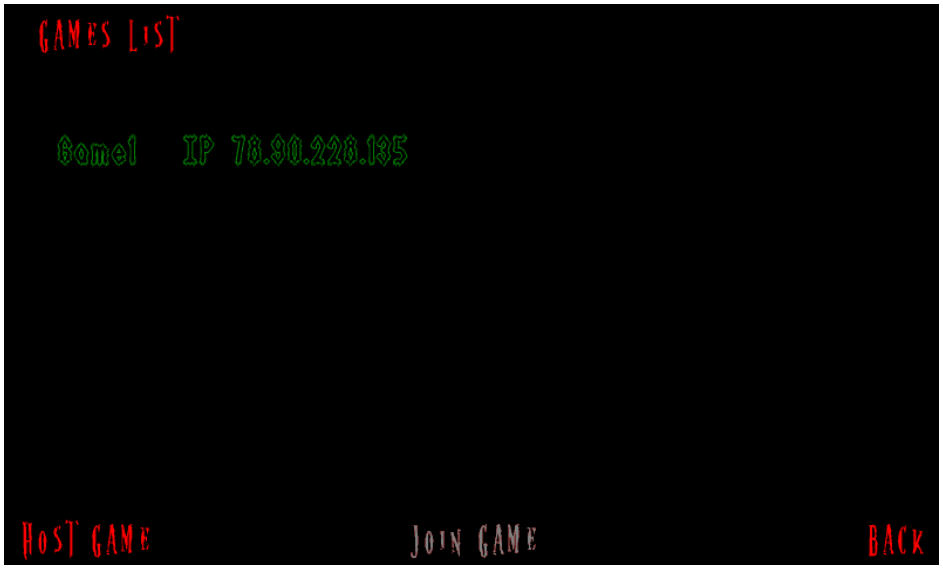
Фиг. 4.3 MainMenu

В началото на играта се пуска т.нар. *Intro*. След него се появява главното меню (фиг 4.3). От него потребителят може да избере четири неща:

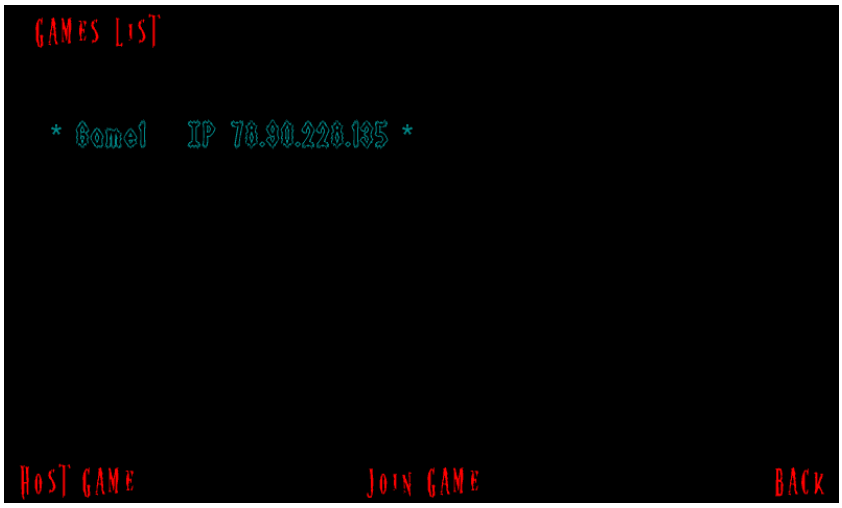
- *Single Player* – влизайки в този режим играчът играе сам срещу зомбитата, управлявани от компютъра.

- *Multiplayer* – този режим позволява на играчът да играе заедно с други хора. Игра се създава чрез бутона *Host Game*. Ако няма игри в мрежата бутона *Join Game* е неактивен, ако има те са показани в списък и играчът трябва да избере една от тях.

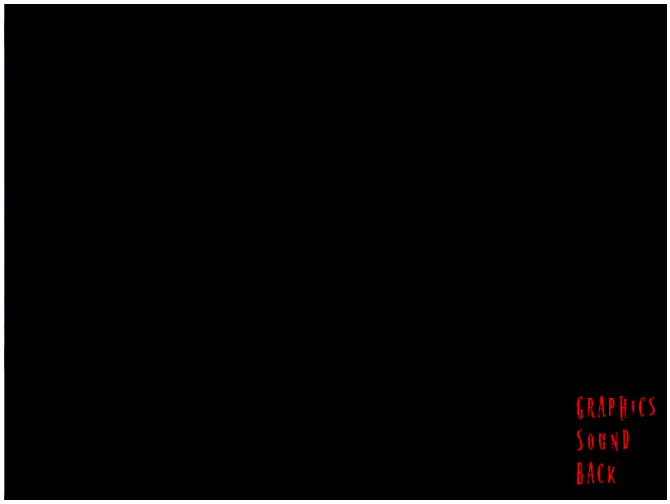
Фиг. 4.4 Multiplayer Menu



Фиг. 4.5 Когато има игра в мрежата тя се показва.

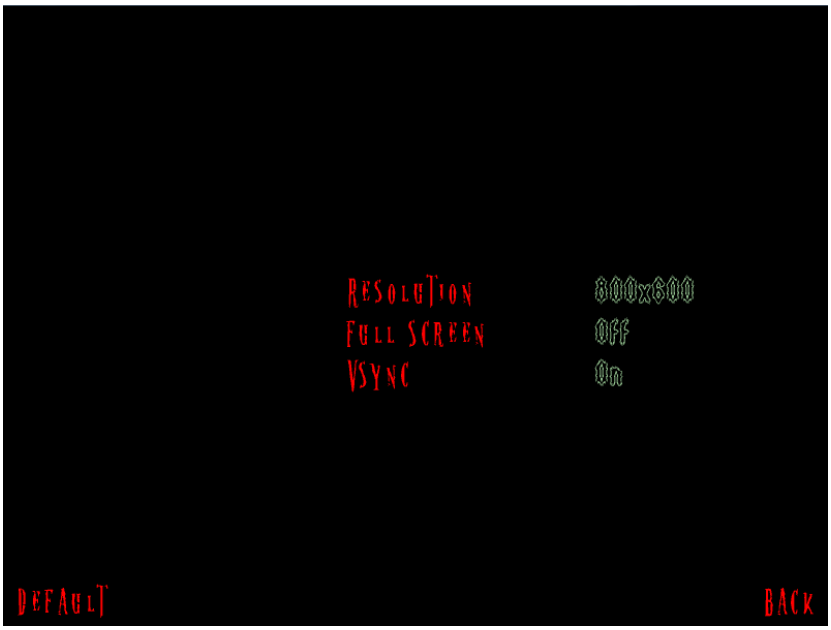
- 

Фиг. 4.6 Избиране на игра.

- *Settings* – препреща играчът в ново меню, където той може да избере:
- 

Фиг. 4.7 Settings Menu

  - *Graphics* – В това меню, потребителят може да избере най-подходящата за него резолюция на екрана, дали играта да се играе в режима *Fullscreen*, който изпълва целият екран. В това меню героят може да избере дали да включи опцията *VSync*.

- 

  Фиг. 4.8 Graphics Menu

- *Sound* – В това меню играчът може да избере нивата на сила на звука според своите лични желания.
- 

  Фиг. 4.9 Sound Menu

- *Exit* – излиза от играта

4.3 Управление

Контролите за управление са:

- W – Напред

- S – Назад
- Мишка – Въртене на камерта
- Десен бутон на мишка - Прицелване
- Ляв бутон на мишка – Стрелба/ удар
- Scroll – Смяна на оръжието
- Esc – Паузиране на играта и връщане в началното меню

Screenshots

Фиг. 4.10 Дърво и къща в далечината.

Фиг. 4.11 Зомбита.

Фиг. 4.12 Нападащо героя зомби.

4.4 Hints

- Револверът има най-голяма сила на удар, но има най-малко амуниции, затова го пазете за „напечени“ ситуации.
- Арбалетът има средна сила на удар, но повече стрели и може да се използва най-често.
- Брадвата е перфектна когато трябва да се разправите с поредното зомби в близък бой. Няма ограничения за ползване.
- По време на играта, зомбитата ви удрят и свалят част от здравето ви. То може да бъде възстановено чрез отвари, които може да откриете в нивата.
- Когато има много зомбита, които ви нападат, най-добрата тактика е бягството, колкото и добър играч да сте – те винаги са повече!

Заключение

С развитието на технологиите се развиват и интерактивните забавления. С еволюцията на видео игрите те ще бъдат по-малко и по-малко асоциирани с *couch potatoes* (хора водещи заседнал начин на живот). Тази еволюция ще направи играенето на игри много по естествено преживяване. Същевременно игрите ще станат по-голяма част от реалния свят. Ще се развиват игри, контролирани от движения. Играчите ще могат не само да контролират героя си с движение на ръцете и краката, но и да предават емоции към вградени герои и други играчи чрез системи за разпознаване на изражението на лицето.

Успешно бе създадена сървайвал хорър игра, която може да достави часове забавление с хора в екип или самостоятелно. С реалистичната си физика, депресираща музика и звуци, зомбита и кръв играта пресъздава типичната атмосфера за този жанр игри. Създаден е и редактор на нива, с който могат да се изграждат най-различни нива. Играта е *open source* и кода ще е изцяло свободен за феновете.

Играта е добре развита, но може да се създаде по-разнообразен *gameplay* като се построят нови нива, анимират други герои и различни зомбита и врагове. Може да се добавят и още оръжия, с които елеминирането на врагове да стане по-вълнуващо, както и да се добавят повече произволни събития в играта, като неочаквано падащи предмети и изненадващо изникващи зомбита.

Използвана литература и ресурси

1) **Beginning XNA 3.0 Game Programming From Novice to Professional** by Alexandre Santos Lobão, Bruno Evangelista, José Antonio Leal de Farias, and Riemer Grootjans
2) **Модели**

   http://www.free3dmodelz.com

   http://www.quality3dmodels.net/categories/architecture-3dmodels.php?page=3

   http://www.turbosquid.com
3) **Инсталатор**

   http://www.clickteam.com/website/usa/install-creator.html
4) **Физичен енджин**

   http://jitter-physics.com/phpBB3/index.php
5) **Звуци и музика**

   http://soundbible.com/tags-horror.html

   http://incompetech.com/m/c/royalty-free/index.html?genre=Horror

Съдържание