ТЕХНОЛОГИЧНО УЧИЛИЩЕ „ЕЛЕКТРОННИ СИСТЕМИ”

към „Технически университет”-София

ДИПЛОМНА РАБОТА

Тема:2D бойна игра на C++

Дипломант:

Асен Стоилов

Научен ръководител:

Мая Милушева

София 2015

УВОД:

Целта на проекта е създаването на двуизмерна бойна игра, която съдържа елементи на жанра „Beat'em up".

Основната идея на играта е да е в ретро стил наподобявайки игрите от този тип от началото на 90-те години. По този начин играта ще може да върне хората назад във времето когато са играли игри като „Cadillacs and Dinosaurs", "Final Fight" и др. на аркадни машини или на конзолата вкъщи.

В момента не съществуват много подобни игри, а портнати оригинални игри от старите модели конзоли и аркадни машини за компютър няма или са непълни с изключение на някои заглавия, която има версия за браузър и мобилно устройство.

1.Информация за играта

1.0. Бойни игри

Бойните игри са вид видео игри,в които се набляга на екшъна, боя, предизвикателствата и бързите реакции. Бойните игри също могат да се разделят на няколко вида-игри с ръкопашен бой, със стрелба, игри със стратегия в реално време и други.

В една игра, независимо дали тя е бойна или не,играчът контролира главния герой като определя движенията му. Главният герой от своя страна може да извършва различни действия-да се бие,да събира елементи,да се бие с врагове. Екшън игрите,както всичко видове игри,най-често са разделени на нива с различни видове трудност. Най-често,играта приключва, когато свърши живота на героя. Играта се играе докато не победиш съперника или докато не изпълниш някаква зададена мисия. Те биват няколко вида:

•Fighting games- Вид бойни игри,които включват бой между герой със специални умения. Играта приключва, когато животът на някои от играчите стане 0. Има 2D и 3D Fighting games, като 3D игрите включват 2D равнина и други специални ефекти. Те се различават от боксовите игри например по това,че движенията и техниките не са толкова реалистични.

•Platform games - Това са тип игри,при които играчите са разположени на платформи, чиито размери най-често са доста преувеличени. Героите се бият върху тези платформи, като избягват разни препятствия.

•Shooter games- Игри,които позволяват на играчът да стреля от разстояние с различни оръжия и да вижда играта от различни ъгли.

1.1. Жанрът “Beat’em up“

,,Beat' em up’’ е вид бойна игра, където героят на играча трябва да се бие с множество врагове в невъоръжена борба или с предмети за такъв тип битка. Играта се състои в минаване през различни нива,където трябва да преминеш определено изпитание за да преминеш в следващото ниво.Накрая на всяко ниво имаш бос,когото трябва да убиеш,за да продължиш в следващото. По някой път тази игра е доста трудна за печелене,което кара играчите да харчат пари,за да се опитат да я спечелят.

,,Beat’em up’’ са свързани с бойните игри, в чиято основа са битките един срещу един и многото врагове. Тази технология е широко разпространена. По времето,когато ,,one-on-one’’ бойните игри и ,,Beat’em up’’ се обединят и имат обща графика и стил,може би ще се появи нов жанр бойни игри.

1.2.История на бойните игри

Началао:

Първата бойна игра е била производство на Sega-боксовата игра ,,Heavyweight’’ шампион (1976), която се разглежда от гледна точка на странично виждане, като по-късностава основа на бойните игри .През 1984г. , ,,Hong Kong cinema’’ прави ,,Kung Fu-Master’’ и така поставя основите на бойните игри със странично виждане с прост геймплей и множество съперници.През същата година излиза играта ,,Bruce Lee’’ ,в която има опции за мултиплейър и избор от няколко герои.По-късно през същата година, ,,Karateka’’ комбинира една по една бойните последователности

„Karate Champ” със свободата на движение в „Kung Fu master”, и то успешно експериментира с добавяне на бойни сцени. Тя е също така и първата бойна игра, която успешно е пригодена към системи за дома. „Nekketsu Kōha Kunio-kun”, излязла през 1986 г. в Япония, се отклонява от бойните изкуства на новите игри и въвежда уличен бой. Западната адаптация „Renegade” добавя подземния свят като част от бойните игри, с което се оказа по-популярна сред геймърите, отколкото на принцип на бойните спортове на другите игри.„Renegade”набира популярност сред бойните игри,защото има възможност героите да се движат хоризонтално и вертикално. Тя въвежда използването на комбо атаки.

“Renegade”-1986г. (аркадна версия)

Златния век:

През 1987, издаването на Double Dragon поставя началото на "златен век" за "Beat 'em up" жанра, който трае почти 5 години. Играта е проектирана да бъде наследник на Renegade, но отвежда жанра до нови висоти с подробен набор от бойни изкуства и кооперативен геймплей. Успеха на Double Dragon довежда до потоп от игри от същия жанр в края на 80-те, където се отличават заглавия като Golden Axe и Final Fight. Final Fight беше предназначена за продължение на Street Fighter, но в последствие Capcom му дава ново заглавие. За разлика от простите комбо атаки в Renegade и Double Dragon, комбо атаки в Final Fight а много по-динамични. Тя е обявена за най-добрата игра в жанра, и изкарва две продължения и по-късно е пренесен към други системи.

“Final Fight” - 1989г.

"Златния век" на жанра свършва с Street Fighter II която връща играчите към битките ено-в-едно, докато нарастващата популярност на 3Д игрите намалила популярността на 2D -базирани боксьорски игри. За края допринася и факта, че до средата да 90-те жанра страда от липса на иновации.

32 битовата ера и натам:

Първата по-интересна игра, излязла след края на „Златния век“ на жанра е „Dynasty Warriors“ и нейните продължения в началото на 2000-те, които въвеждат жанра в третото измерение. Те предлагат традиционния „Beat 'em up“ геймплей, но на големи 3Д бойни полета, показвайки десетки герои едновременно в реално време. Тази поредица заедно с „Ykuza“, остава единствената игра от жанра излязла на пазара до 2002 г.

„Dynasty Warriors 2“

В години след 2004 жанра започва да се преражда и излизат много нови заглавия като: поредиците „*Ninja Gaiden"*(2004-до сега) и *"God of War"*(2005-до сега), както "*Heavenly Sword"(*2007), "*Afro Samurai"*(2009) и "Bayonetta*"*(2009). Дори в последните години са издадени някои нови традиционни 2Д „Beat'em up" игри, като: „Scott Pilgrim vs. the World: The Game "(2010) и поредицата „Shank"(2010-2011).

1.3. Дизайн на игрите „Beat'em up"

Игрите често включват криминални престъпления и отмъстителни битки на улицата, в минали далечни времена на странни места. Играчът може да ходи от единия край на игралния свят до другия и най-често всяко следващо ниво на играта се мести хоризонтално. Някои от по-късните „Beat'em up'' игри с 2D-базирана смяна на нивата позволяват на играчът да извърша толкова много действия, колкото 3D игрите със същия gameplay.

Докато тече самото ниво, играчът може да придобива оръжия и предмети, като някои от тях са такива, че могат примерно да повишават кръвта на героя.

Докато играчът минава през нивото, той е спиран от много врагове, които трябва да победи,преди да продължи напред. Нивото свършва,когато всички врагове са победени. Всяко ниво съдържа много изпитания,което прави превъртането на тезиигри доста трудно. В „Beat'em up'' игрите, накрая на всяко ниво играчите с бият с бос, който е по-силен от всички други врагове.

Този тип игри дават избор на играчът да избере между няколко герои-всеки има специални движения, оръжия и способности. Атаките могат да включват бързи комбинации от атака(комбота). Героите често имат свои специални атаки и

удари,които водят до това играчът да избира различни стратегии как да играе играта. Контролната система е проста за научаване и използване, включваща няколко бутона. Тези бутони могат да бъдат комбинирани за да се осъществяват специални удари и атаки.

След излизането „Double Dragon” се появява нов начин да се играят такъв тип игри, който позволява на двама играчи да играят не един срещу друг,а срещу компютъра. „Beat’em up’’ е повече колективна игра, отколкото другите типове подобни игри.

Пример за колективна игра в „Cadillacs and dinosaurs“

2.Проектиране и развойни средства

2.0 Цели за изпълняване

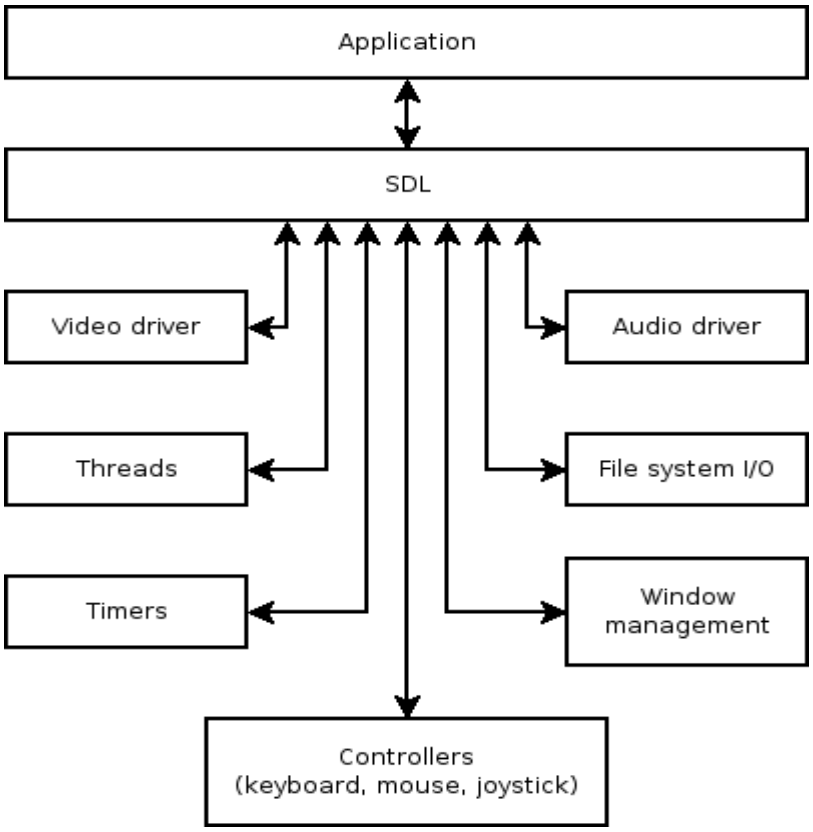
Целта на проекта е създаването на двуизмерна бойна игра, която съдържа елементи на жанра „Beat'em up". Играта трябва да съдържа следните функции.

- Меню от с бутони
- Опция за игра в кооперативен режим и такъв за един човек,
- Две карти на които да може да се играе
- Голям брой различни врагове, които излизат на картата
- Предмети, които могат да се взимат за покачване на нивото на кръвта
- Звукови ефекти на героите и враговете, както и различна музика на всяка една от картите
- Възможност на героя да се движи, тича, скача и да нанася удари благодарение на бойна система с комбинация от удари (комбота)
- Графични елементи като ленти с живот на героите и надписи „You Win" и „You Lose", които да излизат на края на играта.

2.1. Избор на грфична библиотека

За проекта избрах библиотеката SDL2.0, заради нейната добра поддръжка, мултиплатформеността и леснотата с която може да се разработват графични приложения.

Тя работи, като „обвивка“ на драйверите на устройствата свързани към операционната система, както и на нейните функции като многонишковост, управление на прозорци, работа с файловата система и таймерите, като по този начин ни дава лесен достъп към тях, което помага за лесното създаване на проекта.



Архитектурата на SDL.

Като цяло библиотеката е разделена на няколко „субсистеми“:

- **Basics** - занимава се с инициализиране и изключване, обработка на грешки и обработка на входно-изходни операции.
- **Video** – управление на дисплея и прозорците, ускоряване на рендеринга и др.
- **Input Events** - обработване на събития, поддръжка на клавиатура, мишка и игрови контролери.
- **Audio** – пускане и записване на звук
- **Threads** – за работа с многонишковост
- **Timers** – поддръжка на таймери
- **File Abstraction** - за работа с файловата система
- **Power Management –** за статуса на управлението на захранването
- **Platform and CPU Information –** за откриването на платформата и свойствата на процесора
- **Други..**

Освен тези нейни функции SDL за поддръжка на някои други функционалности към нея има и няколко отделни билбиотеки:

- *SDL_image*— поддръжка на повечето формати за изображение

•*SDL_mixer*— комплексни звукови функции главно за миксиране на звуци

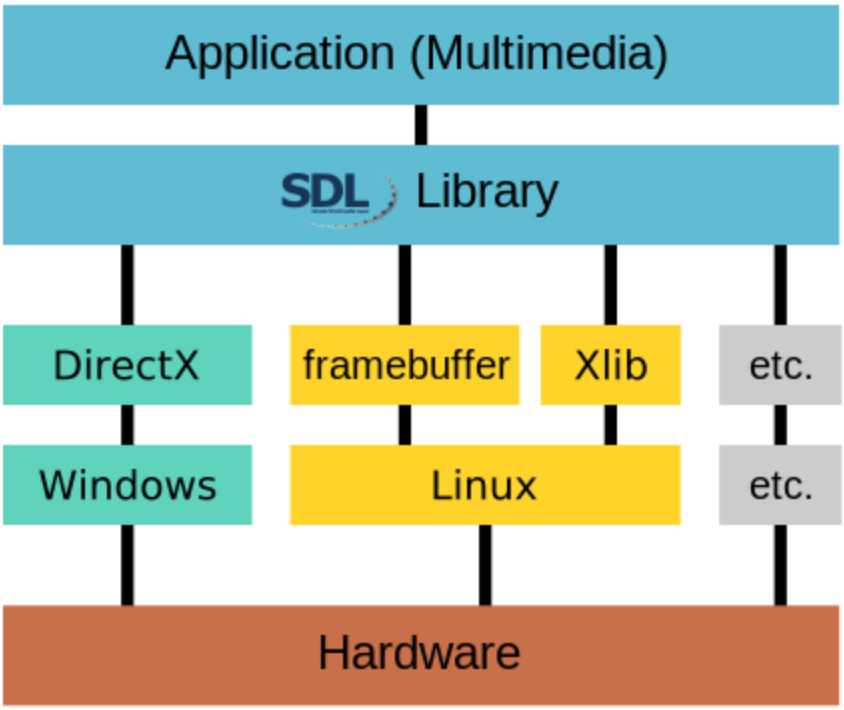
•*SDL_net*— мрежова поддръжка

•*SDL_ttf*—TrueType font rendering support

•*SDL_rtf*— simple Rich Text Format rendering

Библиотеката SDL2.0 може да се използва от много езици като: C, C++, Pascal, Python(viaPySDL2.0), C#, Lua, Ocaml, Vala и Genie.

Заради мултиплатформеността си голяма част от кода на библиотеката е разделен на отделни модули, които са различни за всяка операционна система, като за модула за всяка операционна система се използва различен бакенд, както е показано на долното изображение.



2.2 Избор на програмен език и среда за разработка.

За проекта съм избрал езика за програмиране C++. Причината за избор точно на този програмен език е широката му поддръжка на различни хардуерни и софтуерни платформи. Езиците C/C++ са най-разпространените конвенционални програмни езици. Мултимедийната библиотека SDL има изключително добра поддръжка на тези езици. Тъй като те са проектирани така, че да са максимално близко до хардуера, те гарантират изключително добро бързодействие в сравнение с други езици за програмиране.

Java

N	CPU secs	Elapsed secs	Memory KB	Code B	≈ CPU Load
500,000	0.38	0.38	680	1424	5% 5% 3% 100%
5,000,000	2.42	2.42	23,432	1424	0% 0% 1% 100%
50,000,000	22.67	22.67	23,812	1424	1% 0% 1% 100%

C#

N	CPU secs	Elapsed secs	Memory KB	Code B	≈ CPU Load
500,000	0.32	0.32	888	1305	0% 3% 0% 100%
5,000,000	2.27	2.27	22,272	1305	1% 0% 1% 100%
50,000,000	21.75	21.76	22,304	1305	1% 1% 0% 100%

Ruby

N	CPU secs	Elapsed secs	Memory KB	Code B	≈ CPU Load
500,000	6.58	6.58	7,136	1137	1% 1% 0% 100%
5,000,000	68.62	68.65	7,152	1137	1% 1% 0% 100%
50,000,000	660.09	660.30	7,164	1137	1% 0% 0% 100%

C++

N	CPU secs	Elapsed secs	Memory KB	Code B	≈ CPU Load
500,000	0.10	0.10	?	1544	0% 0% 0% 90%
5,000,000	0.94	0.94	436	1544	0% 1% 0% 100%
50,000,000	9.35	9.35	436	1544	1% 1% 1% 100%

Сравнителна характеристика на езиците при изчисление N-body симулация

За изработка съм използвал Visual Studio 2013 Express, защото е много лесно за работа, безплатно е, има добра поддръжка и помогна за лесна работа с Git системата която ползвах. А за описание на променливите съм използвал JSON, защото е много по-лесен за работа от използвания често в игрите формат XML.

2.3. Описание на алгоритъма

Играта започва в файла main.cpp в него се създава клас GamBase, на който се подават като аргументи дължината и височината на прозореца в който ще се пусне играта. След това се изпълнява метода „init()“ на класа, който създава прозореца и рендерера, както и инициализира звука.

След като се изпълни този метод се създава нов клас GameStateMachine, който ще служи за зареждане излизане и изпълняване на състоянията на играта (ниво, меню и т.н.), след това в него се зарежда състоянието MainMenu, което отговаря за главното меню на играта.

Когато е заредено менюто се пуска главния цикъл на играта, в която се изпълнява SDL_PollEvent(SDL_Event* event) , който служи за отбелязване на висящите събития в момента. Главния цикъл е разделена на три функции

- handleEvent() – отбелязва събитията
- update() - изпълнява действията подтикнати от обработеното събитие.
- render() - рендерира на екрана обектите от моментното състояние.

3. Разработка на проекта

3.0 Използвани структури от SDL.

- SDL_Event – основната структура отговаряща за събитията, от която използвам променливите „type”(за проверка дали има движение на мишката, натиснат е бутон или дали приложението ще бъде изключено) и „key“(за индикиране на натиснатите бутони на клавиатурата).

- SDL_Rect – дефинира правоъгълник, с позиция x и y, както и дължината w и височина h, който помага при изрязването на кадрите на анимацията частта от картата на, която се намира героя.

- SDL_Window – структура, която съдържа цялата информация за прозореца на приложението в себе си(размер, позиция, граници и др.)

- SDL_Rendrer - структура, която манипулира цялото рендериране, тя е свързана със SDL_Window за да може да рендерира само в един определен прозорец.

- Mix_Chunk – съдържа данни за парче аудио, работи само с формата Wav

- Mix_Music – съдържа данните за музикален файл работи cWAV, MOD, MID, OGG, and MP3

3.1. Основни класове

3.1.2.Класът GameBase

Този клас е служи за инициализация на екрана. При създаването си той приема, като аргументи дължината и височината на екрана. Той съдържа структурите SDL_Window, SDL_Renderer и SDL_Event на прозореца на приложението.

Основната функция на този клас е „bool init()“ тя служи за инициализиране на звука, видеото и прозореца на изображението. В него се изпълняват функциите:

- SDL_Init(SDL_INIT_VIDEO | SDL_INIT_AUDIO) - инициализира библиотеката SDL с аудио и видео субсистемите
- SDL_SetHint(SDL_HINT_RENDER_SCALE_QUALITY,"1") - задава филтрирането на текстурите да е линейно
- SDL_CreateWindow("Arcade Fighter", SDL_WINDOWPOS_UNDEFINED, SDL_WINDOWPOS_UNDEFINED windowWidth, windowHeight, SDL_WINDOW_SHOWN) – създава нов прозорец със зададената му дължина, който е видим и може да бъде местен.
- SDL_CreateRenderer(gWindow_, -1, SDL_RENDERER_ACCELERATED|SDL_RENDERER_PRESENTVSYNC) – създава нов SDL_Renderer, синхронизиран с честота на монитора и използва хардуерно ускорение.
- SDL_SetRenderDrawColor(gRenderer_, 0xff, 0xff, 0xff, 0xff) - задава цвета на SDL_Renderer-а на приложението

- (IMG_Init(imgFlags)&IMG_INIT_PNG) - инициализира зареждането на файлове с разширение .png
- Mix_OpenAudio(44100, MIX_DEFAULT_FORMAT, 2, 2048) – инициализиране на апито за звук на SDL с максимална честота 44100Hz, в обикновен формат, на втори канал, с 2048byte за изходен.

3.1.2.Класът GameObject

Това е класа, който е база на всеки обект в приложението структурата му е следната:

```
class GameObject
{
protected:
	int posX_, posY_;
	double mWidth;
	double mHigth;
	SDL_Texture* objTexture;
public:
	bool LoadFromFile(std::string path, SDL_Renderer* gRenderer);
	void free();
	void setColor(Uint8 red, Uint8 green, Uint8 blue);
	void render(int x, int y, SDL_Rect* clip, double angle,SDL_Point*
center, SDL_RendererFlip flip,SDL_Renderer*	gRenderer,int w, int h);
	int getWidth();
	int getHigth();
	int getX();
	int getY();
```

```
};
```

Класът съдържа, текстурата, дължината на обекта, височината и позицията му. Освен това той има функции за зареждане на текстурата, и нейното рендерирането и на екрана .

3.2. State Машината

Служи за лесното разделяне на менюто, нивото и т.н. на отделни състояния, които могат да бъдат разделени на отделни класове за по-структориран код. Машината изглежда по следния начин:

```
class GameState{
protected:
	GameBase *mainGame;
public:
	virtual void update(GameStateMachine *stateMachine) = 0;
	virtual void render(SDL_Renderer* renderer) = 0;
	virtual void handleEvent() = 0;
	virtual bool onEnter(GameBase *mainGame_) = 0;
	virtual bool onExit() = 0;
	virtual std::string getStateID() const = 0;
};
class GameStateMachine{
	std::vector<GameState*> states;
	GameBase *mainGame;
public:
	GameStateMachine(GameBase *mainGame_);
	GameBase* getGameBase();
```

```
        void changeState(GameState *newState);
        void pushState(GameState *newState);
        void popState();
        void handleEvent();
        void update();
        void render(SDL_Renderer* renderer);
};
```

Класът „GameState“ представлява дадено състояние и тези на менюто го наследяват и имплементират неговите виртуални методи. Класът „GameStateMachine“ съхранява тези състояния във „std::vector<GameState*>”, като те могат да бъдат добавяни, премахвани и сменяни от него съответно чрез методите му pushState(), popState() и changeState().

Когато едно състояние е добавено се изпълнява неговата функция onEnter(), която инициализира класа, който го наследява, а при премахване се изпълнява onExit(), правеща обратното.

Методите update(), render() и handleEvent() на „GameStateMachine“, които извикват тези на последното състояние във вектора, които им съответстват по име се изпълняват в главния цикъл на играта и по този начин изпълнява действията на играта.

3.3. Квадратна мрежа за координация

За по лесна координиране на героите в играта е имплементирана мрежа от квадрати. Тя помага с това, че с нея много лесно може да се изчисли, дали даден

враг е пред зад или отстрани на героя, което дава възможност на героя да нанася щети на враговете само когато са в тези позиции.

Мрежата представлява двумерен масив структури Location, които представят квадратчета и съдържат координатите им X и Y. Те се използват като ключове в „std::unordered_map<Location, std::tuple<std::vector<Location*>,int>, LocationHash, Equal> edges“, като информацията към която сочат е съседните квадратчета и неговата цена.

Тези структури се генерират от класа SquareGrid, който също така служи като контейнер за тях.

Най-важната цел на тази мрежа е да помогне за имплементацията на A-Star алгоритъма, който да послужи за да намира пътя на враговете до героя.

3.4. Алгоритъм за намиране на път A-Стар

В основата на този алгоритъм седи алгоритъма на Дийкстра, който работи по следния начин Първоначално всеки връх има безкрайна дължина, която се намалява при изпълнение на алгоритъма. Нека *r* е избраният начален връх, а *c*(i, j)* е теглото на реброто от i до j или безкрайност, ако такова не съществува. Алгоритъмът работи, като поддържа дължината *d(i)* на намерения до момента най-кратък път от *r* до всеки връх *i* и постепенно разширява множество *S* от върховете, за които тази дължина със сигурност е оптимална. Първоначално *d(i)=c*(r, i)*, а *S={r}*. На всяка итерация алгоритъмът избира такъв връх *v*, който не е в *S* и *d(v)* е минимално. Върхът *v* се добавя в *S* и се извършва релаксация: за всеки връх *j*, който не е в *S*,

d(j)=min(d(j), d(v)+c(v, j))*. Алгоритъмът приключва, когато всички върхове на графа са в множеството *S*.

В играта кода на алгоритъма изглежда по следния начин:

```
inline int heuristic(Location a, Location b){
	return abs(a.X - b.X) + abs(a.Y - b.Y);
}
void path_search(SquareGrid grid,Location start, Location goal,
	std::unordered_map<Location, Location, LocationHash, Equal> &came_from,
	std::unordered_map<Location, int, LocationHash, Equal> &cost_so_far){
	PriorityQueue frontier;
	frontier.put(start, 0);
	came_from[start] = start;
	cost_so_far[start] = 0;
	while (!frontier.empty()){
		Location current = frontier.get();
		if (current == goal)
		{
			break;
		}
		for (auto next : grid.neighbors(current))
		{
			int new_cost = cost_so_far[current] + grid.cost(current);
			if (!cost_so_far.count(*next) || new_cost < cost_so_far[*next])
			{
				cost_so_far[*next] = new_cost;
```

```
                int priority = new_cost + heuristic(*next, goal);
                frontier.put(*next, priority);
                came_from[*next] = current;
            }
        }
    }
}
```

Алгоритъма работи по следния начин:

Имаме клас „PriorityQueue frontier”, който служи като стек за проверените квадратчетата заедно със цената за достигането им, като най отгоре на този стек седи квадратчето с най-малка цена за достигане.

Като аргументи са подадени std::unordered_map -ове, единия за индикиране кое квадратче от къде идва и един за цената му за достигане, както и началното и крайното квадратче.

Това което прави алгоритъма е да зарежда в стека от първо началното квадратче сцената му за достигане и след това съседните му докато не свършат и когато свършат се избира квадратчето с най-малка цена за достигане, след което се изпълнява същия процес за него докато не се достигне по този начин до квадратчето което е подадено за крайно.

Това което различава този алгоритъм от този на Дийкстра е че има функцията „heuristic(Location a, Location b)”, която се използва връща разликата от кординатите на квадратчето което се проверява и това което е крайно. Тя се добавя към цената за достигане на квадратчетата, което помага за по точното определяне на цената на пътя.

След като е изпълнен този алгоритъм много лесно може да се реконструира пътя до квадратчето, което е зададено като крайно със „std::unordered_map<Location, Location, LocationHash, Equal> &came_from”, защото след като е запълнен ще връща от къде идва всяко квадратче. Реконструирането става чрез следния код:

```
std::vector<Location> reconstruct_path(Location start, Location goal,
      std::unordered_map<Location, Location, LocationHash, Equal>& came_from)
{
      std::vector<Location> path;
      Location current = goal;
      path.push_back(current);
      do{
            current = came_from[current];
            path.push_back(current);
      } while (current != start);
      return path;
}
```

3.5. Героите в играта

В основата на героите седи класа „GameCharacter“, а основните функции, които притежава са за рендерирането(void renderCharacter(SDL_Renderer* gRenderer)) на героите, за движението им, както и за изчисление на колизии между тях. Класът за играча се нарича „Player“, а за враговете „Enemy“.

Класовете за героите и враговете зареждат своите променливи от JSON, файлове, чийто път е зададен като аргумент при инициализацията им.

Анимациите им се записват в „std::map<std::string, std::vector < SDL_Rect >> animations; “, чийто ключ е стринг, който означава името на анимацията, която ще се изпълнява. Самата анимация представлява vector от структури SDL_Rect, които съдържат размера и координатите на всеки кадър които се подават съответно на функцията render() наследена от класа GameObject, което и задава да рендерира само тази част от заредената текстура, като по този начин при сменянето на кадрите се счздава илюзията, че героя извършва някакво движение.

Текстурата на герой, включен в играта показваща различните кадри от анимацията му.

Действията на героите се извършват в функцията „void update()“, като за героя в нея се те се извършват, когато за тях бъде натиснат съответния бутон, което се разбира от функцията „void handleEvent()“ на която както аргумент се дава структурата

pKeys, която съдържа променливи отговарящи за бутоните за всяко действие. Тази структура се дефинира извън класа, като всяка променлива трябва да съответства на идентификатора на всеки бутон.

Натиснатите бутони се разбират чрез функцията SDL_GetKeyboardState(NULL), която връща масив от идентификаторите на натиснатите бутони, като при сравняване със елементите на структурата зададена, като аргумент се разбита дали ще се извърши дадено действие.

Героя може да извършва множество от действия, както и да взаимодейства с враговете и предметите около него. Алгоритъма за действията му изглежда по следния начин:

```
bool Player::actions(){
	if (playerEvent.superPunch || sPunch){
		superPunch();
		return true;
	}
	if (playerEvent.runPunch || rPunch){
		runPunch();
		return true;
	}
	if (playerEvent.jump || jumping){
		jump();
		return true;
	}
	if (playerEvent.grab || grabing){
```

```
                grab();
                return true;
        }
        if (playerEvent.normalPunch){
                punch();
                return true;
        }
        if (currentCondition == PUNCHED){
                fall();
                return true;
        }
        return false;
}
void Player::moving(){
        if (playerEvent.moveRight && !playerEvent.normalPunch){
                moveRight();
        }
        else if (playerEvent.moveLeft && !playerEvent.normalPunch){
                moveLeft();
        }
        else if (playerEvent.moveUP && !jumping){
                moveUp();
        }
        else if (playerEvent.moveDown && !jumping){
                moveDown();
        }
}
```

При враговете действията се извършват от сления алгоритъм:

```
void Enemy1::update(std::list<GameCharacter*> characters)
{
	if (health > 0){
		doActions(characters);
	}
	else{
		if (target_ != NULL){
			std::get<AVAILABLE>(players[playernum]) = true;
			target_ = NULL;
		}
		framesToEnd--;
		animation("FALLING");
		frame =4* 4;
	}
	resizeClips(&Clips[frame/4]);
}
```

Той работи по следния начин, проверява дали кръвта на героя е по-голяма от нула и ако е, ако отбелязал някой играч, като цел я маха и слага кадъра в който е паднал на земята, да се рендерира на екрана, докато не минат определен брой кадри за да бъде премахнат от екрана.

Ако героя е с кръв по-голяма от 0 то се изпълнява следната функция:

```
void Enemy1::doActions(std::list<GameCharacter*> characters){
	punched = false;
	moveDir = { true, true, true, true };
	collision(characters);
```

```
	currentCondition = STANDING;
	action = false;
	currentGoal = getGoalSquare();

	if (player_punching()){
		fall();
	}
	else{
		animation("WALLKING");
		if (currentGoal != currentSquare[0]){
			moving();
		}
		if (action){
			frame++;
		}
		else punch_players();
	}
	if (frame / 4 >= Clips.size()){
		frame = firstclip;
	}
}
```

Тя работи по следения начин:

1. изпълнява функцията извършваща намирането на колизии
2. задава цел на алгоритъма на за намиране на път и ако тя не е играча целта става квадрата на който се намира врага.

3. Проверява дали героя нанася удари по него и ако е така изпълнява анимацията за падане на земята.

4. Проверява дали е достигнал до целта си и ако не так продължава да върви към нея, ако не започва да нанася удари на героя

5. сменя кадъра номера на кадъра

3.6.Състояния

В играта има две състояния:

MainMenu – това е състоянието на менюто. В неговата основа седи „std::map<int, GameButton*> button “, служи за съхранение на класовете на бутоните, като всеки от тях има ключ който да отговаря за него. Аргументите за тях заедно с тези на фона се зареждат от JSON файла „Resources/menuData.json“.

Менюто е разделено на три състояния:

- Menu1-в него се извежда на екрана бутон за стартиране на играта
- Menu2-в него се извеждат и извършват действията за бутоните отговарящи за избирането дали режима на игра е за едни или двама човека, като това се записва в променливата „gameMode“.
- Menu3-в него се извеждат бутоните за нивата, и при натискане на някои от тези бутони се създава ниво съответно на натиснатия бутон с аргумент индикиращ режима на игра от предишното състояние.

Състоянията могат да се връщат назад със натискане на бутона, което е имплементирано във функцията „handleEven()“, чрез следния код:

```
void MainMenu::handleEvent()

{

	switch (mainGame->gameEvent.type){

	case SDL_KEYUP:

		if (mainGame->gameEvent.key.keysym.sym == SDLK_ESCAPE){

			esckapePressed = true;

		}

		break;

	case SDL_KEYDOWN:

		if (mainGame->gameEvent.key.keysym.sym == SDLK_ESCAPE &&

			esckapePressed){

			esckapePressed = false;

		if (currentMenu > 0) currentMenu--;

			else mainGame->quit = true;

		}

		break;

	}

}
```

GameLevel – класа на състоянието отговарящо за менюто. При “пушването” си във вектора на класа GameStateMachine той зарежда променливите нужни за класовете и ги създава на враговете, героите и изображенията които ще бъдат заредени на екрана от JSON файла „Resources/level.json“. В него е описани:

- врагове, които ще са в нивото

- променливите за лентите за кръв на героя/ите на играча или ииграчите респективно.
- пътя към JSON файла, описващ героите
- текстурата за фона на нивото

Героите на играча се добавят в „std::vector < std::tuple<GameCharacter*, bool> > players “, като клас на героя е в std::tuple с променлива от типа bool, която индикира, дали героя е взет за цел от някой от враговете.

Героите, които в дадения момент на игра извършват действия нивото са добавят в std::list-a characterList. Във функциите „update“ и „render“ елемент по елемент се итерират класовете на героите и се изпълняват съответните функции на тези в които са. Когато някои от героите има кръв раван на нула той се изтрива от този списък, а ако той е този на играча играта приключва.

Самото ниво е разделено на позиции на камерата, които представляват отместването на нивото надясно с една пета от дължината на фона на нивото. По този начин се симулира напредването на героя по картата.

Враговете в нивото се създават в отделни std::list-ове, като за всяка позиция на камерата на нивото има по един такъв лист, като всичките те са във вектора „enemies“.

Съответстващите списъци на враговете съответстващи с позициите на камерата се добавят към главния, когато те се сменят както е показно на долния код:

```
void GameLevel::manage_camera(){
      if (cameraPosCount == CAMERA_POSITIONS-1 && livingEnemies <= 0)
```

```
	{
		currentState = WIN;
	}
	if (cameraPosCount < CAMERA_POSITIONS){
		int playersAtEndOfCAmera = 0;
		for (auto player : players_){
			//last x position on screen = 90% from screen W
			if (player->getX() > mainGame->getScreenW()*(0.90))
			{
				playersAtEndOfCAmera++;
			}
		}
		if (playersAtEndOfCAmera == players.size() && livingEnemies<=0){
			for (auto player : players_) player->manageCameraPos();
			camera.x +=backGroundLevel1->getWidth()/5;
			cameraPosCount++;
			itm = (*items)[cameraPosCount];
			livingEnemies = enemies[cameraPosCount].size();
			charactersList.insert(charactersList.end(),
			enemies[cameraPosCount].begin(),enemies[cameraPosCount].end());
		}
	}
}
```

Както е показано камерата се сменя когато позицията X на играчите е по голяма от 90% от дължината на екрана, когато е достигната последната

Освен споменатите функции класът има и следните по-главни функции:

- enemy_add_target() - задава като цели за враговете героите героите

- LoadObjects() −зарежда текстурите на обектите и музиката съответстваща на нивата.
- DrawPlayerHealthBar() - изкарва на екрана лентите с кръвта на играчите

Състоянието на нивото също като това на менюто е разделено на три под-състояния:

- INGAME – когато играчите и враговете са с кръв над нула
- GAME_OVER – когато някой от играчите е с кръв под нула.
- WIN – когато играча или играчите са спечелили играта.

3.7. Други обекти в играта

Освен споменатите класове, които са в основата на играта има още няколко които придават завършен вид на играта. Те са:

GameButton: това е класът, който имплементира бутоните в играта. В основата му седи структура SDL_Rect, която съдържа позицията и размера на бутона. Чрез функцията „SDL_GetMouseState(int *x, int *y)" се взема позицията на мишката, която се сравнява с тази на бутона и когато е върху него и е или не е натиснат бутон на мишката , се сменя клипът на текстурата и ако е натиснат се връща състояние „true"

GameLabel: представлява изображение, което може да бъде рендерирано на екрана в зададена позиция.

BackGround: служи за фон на изображението и може да работи със смяна на позициите на камерата.

HealthBar: това е класът на лентата на героя. Той представлява изображение, което е задната част на лентата и червен правоъгълник, който се рисува върху него и се намалява спрямо количеството кръв.

Item: представлява GameLabel, който служи, като предмет и има стойност за кръвта която връща на героя, когато бива взет.

Items: специален стек за предметите.

4.Ръководство за потребителя

4.1. Изисквания за играта

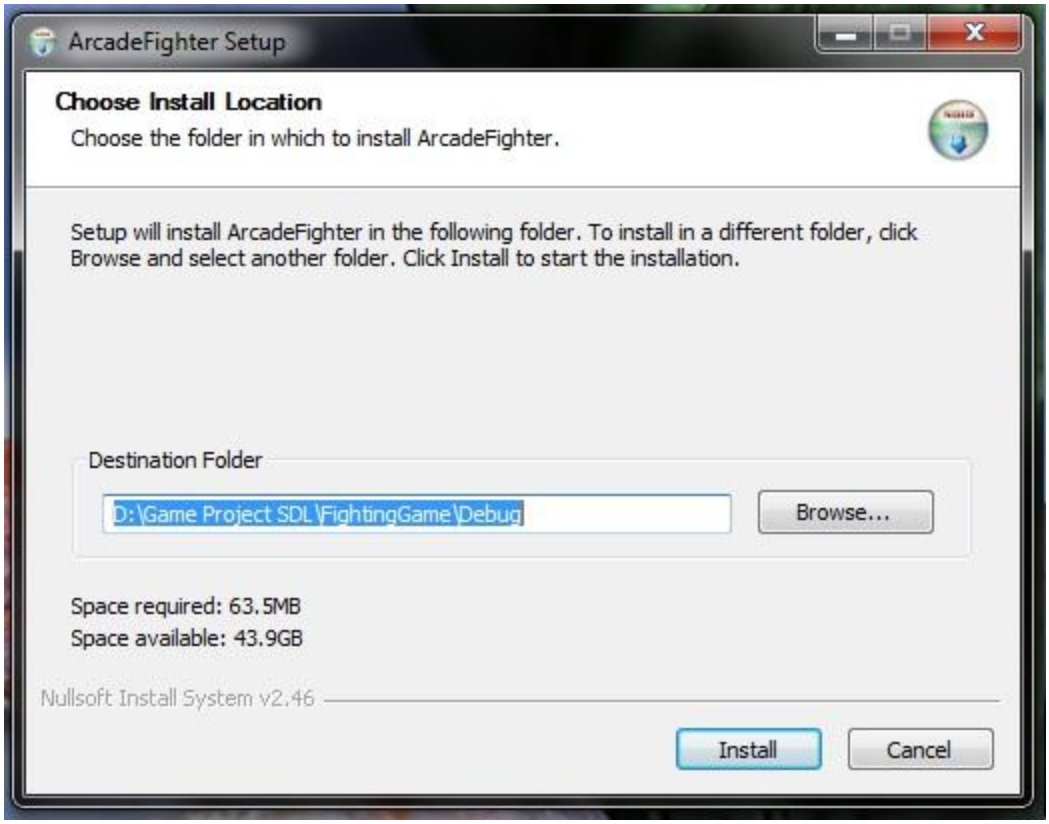
Изискванията за играта от към хардуер са прости играта би трябвало да върви на всеки компютър със параметри по-високи от следни:

- Процесор: 1.5GHz или по-добър.
- Оперативна памет: 256 MB.
- Дисково пространство: Поне 100 MB.
- Налични клавиатура и мишка.

От към софтуер играта изисква операционна система Windows XP или по нова.

4.2 Инсталиране

Инсталирането на играта става от файла „ArcadeFighterInstaller.exe“.



Инсталатора на играта.

Единственото действие, което трябва да се извърши е да се зададе папка за инсталиране на играта и да се натисне бутона „install“.

Играта се стартира от файла „ArcadeFighter.exe“.

4.3 Стартиране на играта

Когато се пусне играта излиза следното меню:

Началното меню на играта

За да започне играта се натиска бутона старт след което излизат следните бутони:

Като бутон „1p Mode” е за режим за един човек, а „2p Mode” за двама. След като е

избран режима излизат тези бутони:

Те служат за избиране на ниво на което да се играе.

Възможно е връщане назад с бутона „Esc“.

4.4 Начин на игра

Основната логика на играта е следната:

Снимка от играта

Имам един или двама герои които излизат от лявата страна на картата, а от дясната страна излизат врагове които могат да бъдат пребити, а на земята на произволен принцип са генерирани предмети, които представляват вид храна, която връща кръв на героя, който я е взел.

Дадените герои могат да правят следните неща:

- да се движат наляво, надясно, нагоре или надолу - при първия герой със бутоните W, A ,S и D на клавиатурата, а втория със стрелките
- да удрят – при играч1 със бутона V, а играч2 с Ctrl
- да тичат – при играч1 със бутона B, а играч2 с LShift
- да скачат – при играч1 със бутона N, а играч2 с Alt
- да взимат предмет – при играч1 със бутона G, а играч2 с Enter

Възможно е да се правят комбинации от удари като те са два вида:

- въртящ се удар- прави се при натискането на бутона за удар и след това бутона за взимане. Той нанася двойни щети на враговете в сравнение с обикновения удар.

„летящ“ удар – прави се при натискането на бутона за удряне, последван от този за тичане. Той нанася тройни щети и дава възможността на героя да се предвижи бързо до врага, чрез натискане на бутоните за движение.

5. Заключение

Всички първоначално поставени цели за разработване са напълно изпълнени. Това включва – разработка на двуизмерна развлекателна мултимедийна игра, използване на мултимедийната библиотека SDL. Имплементирана е бойна система заедно с две карти за игра и голям брой различни врагове.

Основните проблеми решени в проекта са: рендерирането на обектите на екрана, имплементиране на алгоритъма за намиране на път на враговете, бойната система и създаването на система за колизии. Освен тези проблеми са имплементирани и много други решения като: Системата за сменяне на състоянията на играта, звуковите ефекти на играта и др.

Като бъдещо развитие на проекта може да се направи режим за LAN multyplayer режим, и добавяне на по сложни нива с повече врагове.

Информационни източници:

- http://lazyfoo.net/tutorials/SDL/index.php
- http://www.redblobgames.com/pathfinding/a-star/implementation.html#cplusplus
- http://www.spriters-resource.com/arcade/
- “SDL Game Development” - Shaun Ross Mitchell
- http://www.cplusplus.com/
- https://www.libsdl.org
- http://en.wikipedia.org/wiki/Beat_%27em_up
- http://en.wikipedia.org/wiki/Fighting_game
- http://shootout.alioth.debian.org/u64/which-programming-languages-are-fastest.php

Съдържание