**ТЕХНОЛОГИЧНО УЧИЛИЩЕ ЕЛЕКТРОННИ СИСТЕМИ**

**към ТЕХНИЧЕСКИ УНИВЕРСИТЕТ - СОФИЯ**

**ДИПЛОМНА РАБОТА**

Тема: 3D survival игра използваща Constructive Solid Geometry

Дипломант: Научен ръководител:

Петър ЦанковЕмил Гоцев

СОФИЯ

2017

**TЕХНОЛОГИЧНО УЧИЛИЩЕ**



**ЕЛЕКТРОННИ СИСТЕМИ**

**към ТЕХНИЧЕСКИ УНИВЕРСИТЕТ - СОФИЯ**

Дата на заданието: 10.11.2016 г. Утвърждавам:..............................

Дата на предаване: 10.02.2017 г. /проф. д-р инж. Т. Василева/

**ЗАДАНИЕ**

**за дипломна работа**

на ученика Петър Цанков Цанков 12 Б клас

1.Тема: 3D survival игра на Java използваща Constructive Solid Geometry

2.Изисквания:

Играчът трябва да:

1. има възможност да променя терена, да добавя нови типове обекти и текстури;

2. избира в кой свят да влезе от главното меню;

3. има инвентар с максимален обем и максимална тежест;

4. има възможност да определя как да се генерират върху терена обектите, които вече е добавил - размери, плътност, височина и с каква честота.

3.Съдържание 3.1 Обзор

3.2 Същинска част

3.3 Приложение

Дипломант :...........................................

Ръководител:..........................................

/Емил Гоцев/

Директор:................................................

/ доц. д-р инж. Ст. Стефанова /

**Увод**

Игрите са най-разпространените средства за развлечение в модерния свят.

Разработката на компютърни игри е важна част от индустрията на информационните технологии. Историята на компютърните игри започва от 50-те години на XX век. Първата игра създадена за компютър е била симулация на шах.

За да се създаде една игра са нужни много различни компоненти.

Един компонент, който е ненужен, но полезен, е енджина. Това представлява софтуер, чрез който може да бъде създадена една игра без да бъде писан код на ниско ниво (при някои енджини дори не трябва да се пише код, за да се направи игра). Енджините обикновено предоставят възможност за зареждане на модели, управлението им, както и специфични за жанра на играта възможности. Важен елемент за създаване на игра е да бъдат създадени моделите и текстурите. Освен това трябва да бъде написана логиката как тези модели да си взаимодействат и какво да се случи при взаимодействието им.

Всяко следващо поколение игри могат да бъдат все по реалистични, поради еволюирането на хардуера и начините за оптимизация на софтуера. Разработват се алгоритми, които не само генерират терен, но и npc-та, тяхното поведение и звуци. Целта на дипломната работа е да бъде създадена триизмерна игра, използваща OpenGL през Java. Тя ще е съвместима с Windows, Mac и Linux. Играта ще бъде single-player survival sandbox.

Първа глава

Преглед на подобни игри и средства за създаването им

* 1. **Преглед на подобни игри**
     1. **Space Engineers**

Space Engineers е single player и multiplayer sandbox игра за строене, изследване и оцеляване в космоса и на планети. В тази игра играчът може да строи космически кораби, космически станции и бази на планети. Играта е пусната за Steam през 2013 година. Създателите и са я замислили още през 2002 година, но тогава технологиите за видео игри не са били достатъчно добри за създаването на такава игра, поради което е била създадена по-малка игра “Miner Wars 2081”.

По начало играта започва в космоса, като играчът има кораб с определен размер според трудността, която е избрал за света.



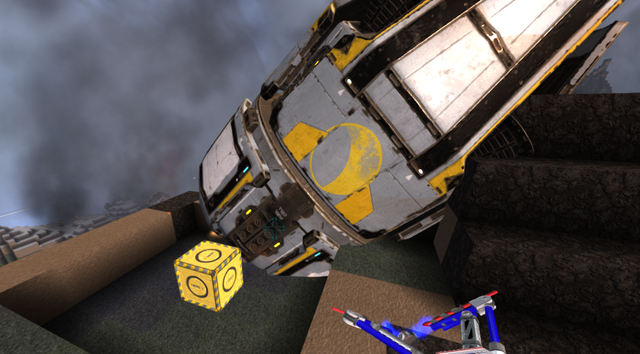
**Фиг. 1.1 начален кораб**

Играчът може да се развива като използва ресурси от астероидите, които се виждат наоколо. В играта може да използваш части за строене на бази или кораби, които могат да се направят от налични ресурси или като разглобиш свои минали творения. Съществуват пирати, които могат по всяко време да те нападнат и биха могли да унищожат всичко, което имаш, но ако ги победиш можеш да използваш техните кораби за резервни части. В играта има и кораби дронове, които можеш да контролираш от разстояние, без да влизаш в тях. Играта няма крайна цел, но вместо това предоставя безкрайни възможни развръзки.

**1.1.2 FortressCraft Evolved**

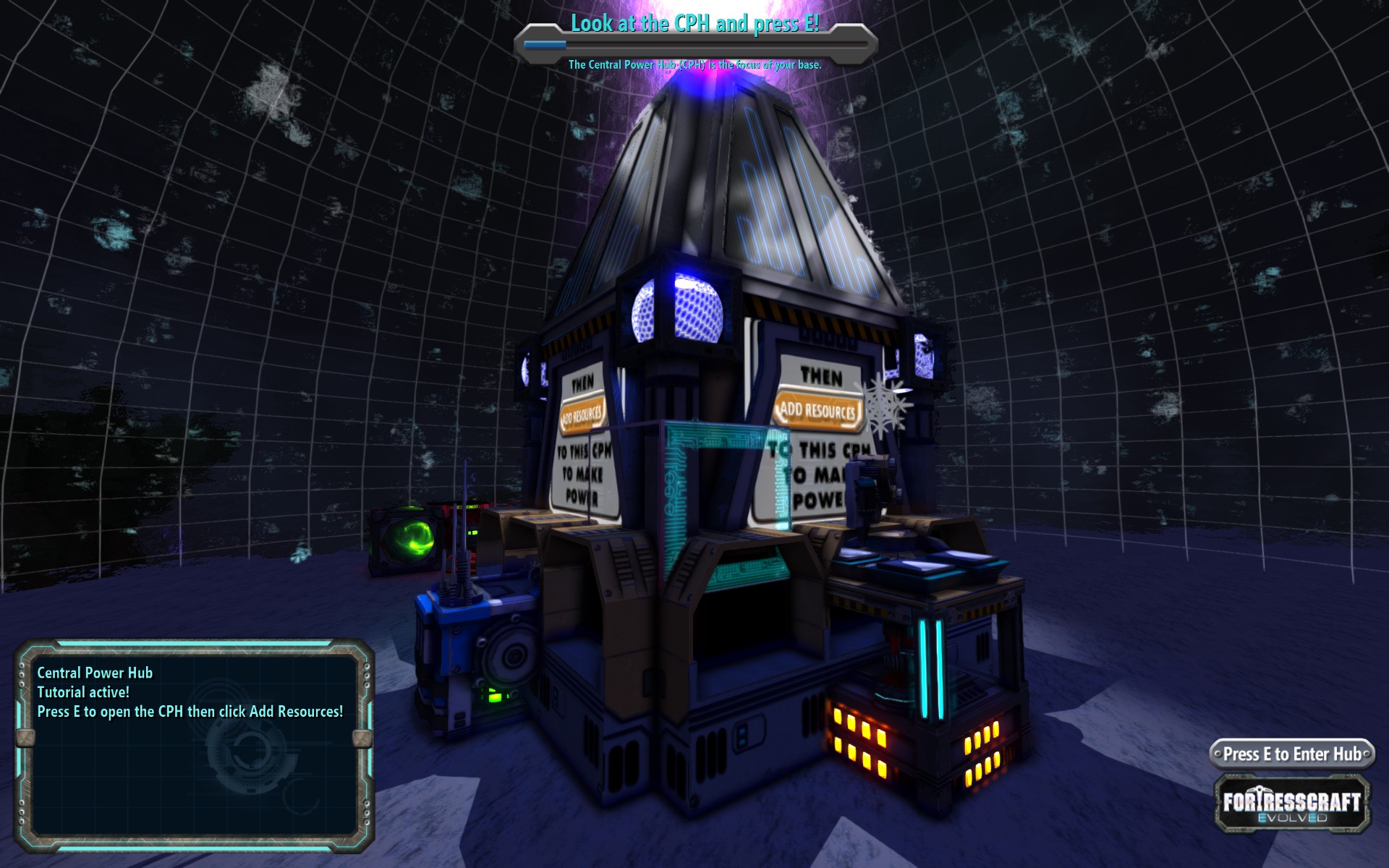
FortressCraft Evolved е 3D sandbox singleplayer/multiplayer игра, която е базирана на воксели (елементи в триизмерен масив, представляващи материя). Играта е направена по подобие на оригиналната игра FortressCraft. И двете игри са създадени от един човек. FortressCraft Evolved съчетава Tower defence, Combat и Sandbox.

Играта започва след като твоят кораб се разбие на извънземна планета.



**Фиг. 1.2 парче от разбития кораб.**

Най-важната част в играта е главното ти ядро (Central Power Hub). Ако то бъде разрушено, според избраната трудност на света или ще трябва да го поправиш (casual), или света се изтрива (hardcore).



**Фиг. 1.3 CPH (Central Power Hub)**

От време на време се случва базата ти да бъде нападната от извънземните. Когато това се случи трябва да защитиш ядрото, като можеш да атакуваш чудовищата с оръжия от играча или да оставиш базата ти сама да се защити, използвайки оръдейни кули.

В играта освен това има руди в земята, с които можеш да направиш по-добри батерии, по-добри машини и по-силни оръжия.

Крайната цел на играта е да се направи кула за предаване на енергия в космоса. Ако играчът успее да предаде един милион енергия, без неговото ядро да бъде унищожено, той печели играта, но играта може да се играе и след това.

**1.1.3 Minecraft**

Minecraft е 3D sandbox singleplayer/multiplayer игра. Играта, въпреки че няма край, си има early-game, mid-game и end-game. В началото на играта играчът е поставен в един произволно генериран свят, без нищо в себе си и трябва да събере ресурси и да построи някакъв подслон преди слънцето да залезе.

През нощта и в тъмни пространства се появяват чудовища, някои от които са:

зомби, скелет, крийпър и паяк.

Зомбитата и скелетите горят през деня, а паяците спират да атакуват героя.

Крийпъра експлодира, ако стои достатъчно дълго близо до теб и по този начин е едно от малкото чудовища в играта, които могат да разрушават терен.

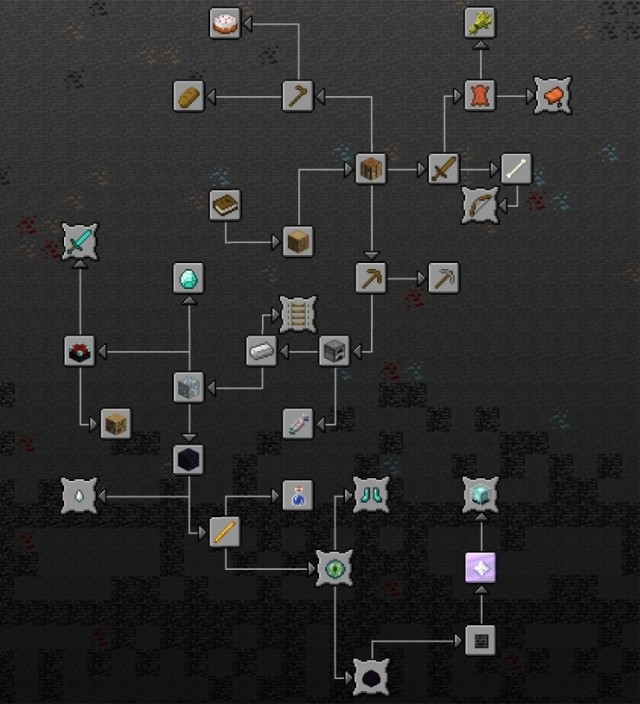


**Фиг. 1.4 зомби, скелет, кръйпър, паяк**

Когато играчът стигне mid-game той може да влезе в ада, където за да продължи да прави прогрес в играта трябва да намери крепости и да убие чудовище приличащо на жив огън.

Към end-game играчът може да отиде в измерението наречено „the end”. В тази част на играта има Бос, който ако го убиеш създава портал, чрез който можеш да се върнеш в нормалния свят.

По време на играта играчът бива направляван от achievements.



**Фиг. 1.5 achievements.**

Minecraft има механики, с които един човек може да направи машина за добавяне на minigame или допълнителни механики към играта, без да трябва да модифицира самата игра.

Например в самата игра могат да се пишат единични команди, които да си взаимодействат.

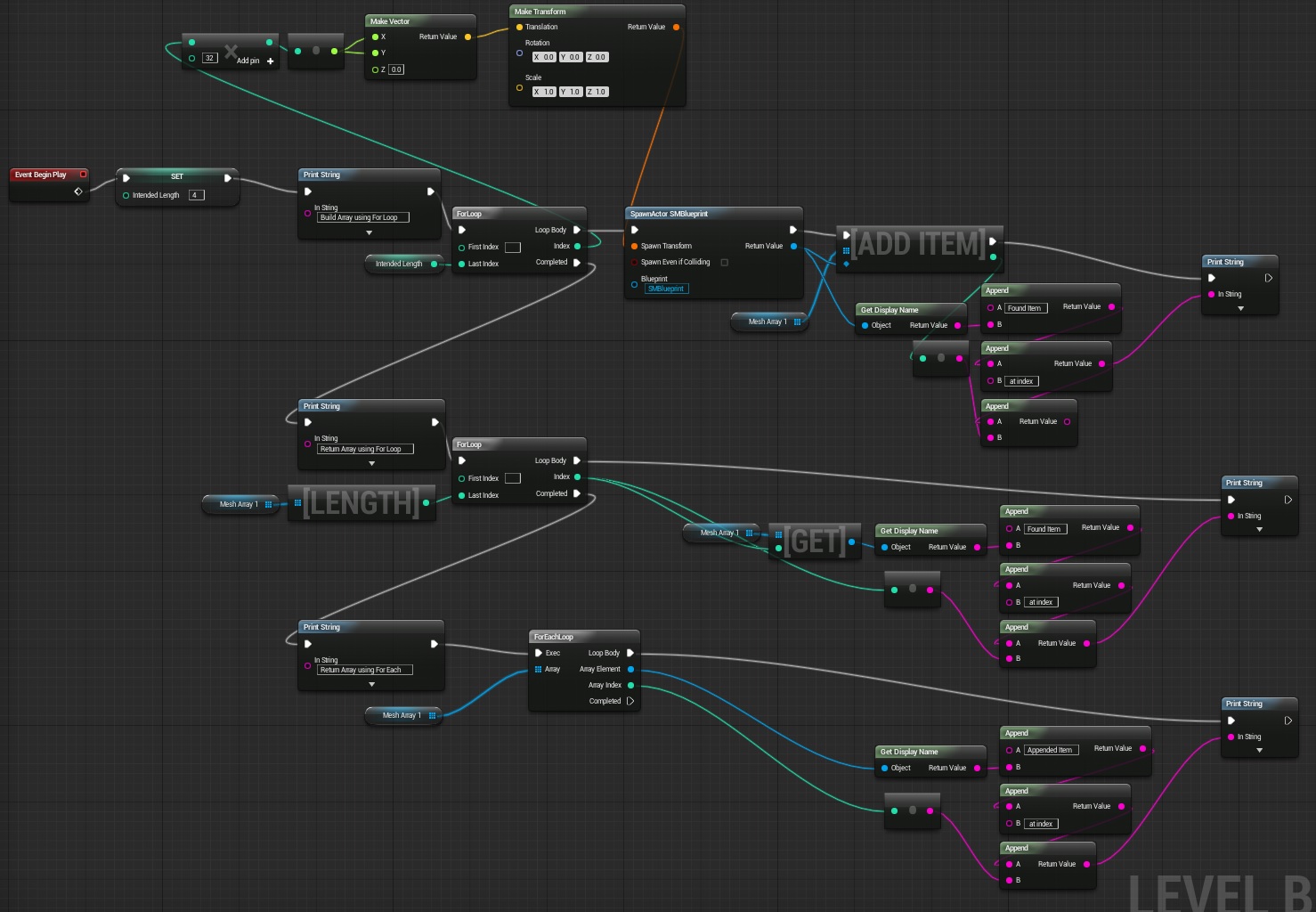
**1.2 Преглед на известните средства за създаване на игри.**

**1.2.1 Unreal engine 4**

Unreal Engine 4 е безплатен гейм енджин, на който можеш да направиш всякакъв вид игра (2D и 3D). Чрез Unreal можеш да направиш игра, използвайки Blueprint, нещо подобно на блок схема, което наричат визуално скриптиране. Unreal създават Blueprint за да може хора да създават игри без да трябва да се занимават с научаването на език за програмиране или без изобщо да научават как да програмират.

В Unreal могат да се създават игри и чрез C++.

Но Blueprint може да бъде преведен на C++ код.



**Фиг. 1.6 Blueprint.**

Ако използвате Unreal Engine 4 за да направите нещо, от което изкарвате пари, вие трябва да дадете 5% от спечелените си пари на EpicGames (създателите на Unreal Engine).

**1.2.2 OpenGL**

OpenGL е библиотека за 2D и 3D графика.

OpenGL е използвана от почти всички операционни системи, за да не се използват различни библиотеки за графика на различни операционни системи.

Библиотеката може да се използва от Ada, C, C++, Fortran, Python, Perl и Java.

OpenGl е пусната за употреба през 1992 година.

OpenGl e безплатна библиотека, разработена от CGI (Silicon Graphics Incorporated).

**Втора Глава**

**Изисквания, средства за създаване на играта**

**и използвани алгоритми**

**2.1 Изисквания към играта**

Играта трябва да съдържа следните възможности:

* Да може да генерира терен динамично
* Да може играча да премахва и добавя към терена с ляв и десен бутон на мишката.
* Да има различни видове обекти като част от терена
* Да симулира камера, която може да се мести с бутоните „w” „a” „s” „d”

и мишката

* Да може да се променя типа на обект, който да се поставя с бутоните 1 и 2

**2.2 Избор на средства за създаване на играта**

За език за програмиране е избрана Java, тъй като Java проект може да се изпълни на много различни операционни системи без да се прекомпилира.

Придобитите знания по Java в 11 клас и писането на модификации за minecraft улесниха създаването на проекта, тъй като не трябваше да се научава нов език за програмиране за да се направи проекта.

За IDE е използван NetBeans IDE 8.2., тъй като тази среда има всичко нужно за създаване на един Java проект, менютата са по-добре организирани от Eclipse и графичният му интерфейс е по-удобен.

**2.3 Описание на алгоритъма**

Приложението започва от функцията main() в класа Main.

Създава се инстанция на логиката на играта и на класа GameEngine.

В конструктора на този обект се създава прозореца, в който ще се изобразява съдържанието на играта.

След това във функцията main() се извиква функцията start() на класа GameEngine. В тази функция според операционната система се извиква функцията run() или функцията start() на нишката за игровия цикъл.

След това започва да се изпълнява главната част - логиката на играта.

В логиката (DummyGame) се създава света, който се показва. Освен това се ъпдейтва света, рендерира се света и обработка на входните данни от потребителя (клавиатура и мишка).

Ако се натисне Esc или X на прозореца, прозореца се премахва, и след това се освобождава паметта, която програмата използва за графични обекти.

Светът, който се рендерира, има в себе си HashMap, където се запазват обекти от тип chunk. Всеки chunk *[3.2.3.3]* има собствен Mesh (Java обект, който улеснява ползването на OpenGl буфери). Това е обект, който направо се показва от GPU-то. Тези Mesh-ове са създадени от 3D обекти, които са запазени в chunk-a.

**Трета Глава**

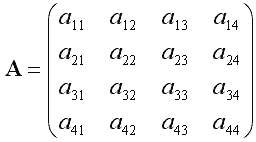
**Реализация на триизмерната игра**

**3.1 Основни понятия в игрите.**

Едно основно понятие в 3D игрите са вертекси. Те представляват точки в 3D пространството, които могат да носят различна информация за себе си, като цвят или координата на текстура. Чрез вертекси се описват всички обекти, които се рисуват на екрана. Друго основно понятие от игрите са моделите (meshes). Моделите представляват съвкупност от вертекси с различни позиции. Тези вертекси са групирани по тройки, като всеки 3 създават едно лице, което GPU-то може да покаже на екрана. Няколко програми за създаване на модели са Blender и Autodesk 3ds max. За да се използва един модел в игра, този модел трябва да се създаде чрез програма и да се експортира в нужният ни за програмата формат или бихме могли да създадем ръчно малки обекти, чрез описване на точките и лицата им.

След това моделът се зарежда в играта като се четат стойностите на всеки вертекс и се записват във vertex и index buffer.Vertex буферът съдържа стойностите на всеки вертекс, а index буферът съдържа индекси, които сочат към вертекс от Vertex буфера. Ако няколко индекса сочат към един и същ вертекс, то той ще бъде нарисуван само веднъж, като без index буфер би се прерисувал няколко пъти. Използването на index буфера се налага, тъй като вертексите се организират в триъгълници и при рисуването им някои вертекси се припокриват и следователно се прерисуват няколко пъти. При не особено големи модели (големи в смисъл да имат голям брой вертекси) това не е проблем, но при рисуване на голям терен може да предизвика спад в производителността на играта.

Важна част от игрите са матриците. Те съдържат колони и редове.



**Фиг. 3.1 матрица с 4 реда и 4 колони**

Матриците се използват за да се преместят, завъртят или увеличат дадени обекти.

По този начин може даден обект да има local space и все пак да можем да го покажем в world space.

По този начин може да се симулира операцията на местеща се камера в пространството, тъй като да се изчисли как ще изглежда света от друга гледна точка би било твърде не ефективно, се използва матрица, с която да се преместят и завъртят всички обекти, което да прилича на местене на героя в пространството.

В OpenGL това става, като подадеш на vertexshader-а *[3.2.2]* матрица, с която да обработиш координатите на всеки вертекс.

**3.2 Описване на начина на реализация**

За реализация на играта е необходимо да се реализират следните етапи:

-Написване на енджин за управление на играта

-Създаване гейм логиката

-Написване на шейдъри за визуализиране на съдържанието в играта

-Създаване на модели, които да бъдат ползвани в играта

**3.2.1 Енджина**

GameEngine в нашият случай е thread, който контролира времето между отделните функции, които енджина поддържа, като:

-Времето между всеки frame и дали да се пропусне даден frame, ако програмата се забавя.

-Ъпдейтването на играта според това колко време е минало между сега и миналото ъпдейтване.

-Взимане на input от играча и предаване към логиката за обработка.

Това се случва във функцията на фигура 3.2.

protected void gameLoop() {

float elapsedTime;

float accumulator = 0f;

float interval = 1f / TARGET\_UPS;

boolean running = true;

while (running && !window.windowShouldClose()) {

elapsedTime = timer.getElapsedTime();

accumulator += elapsedTime;

input();

while (accumulator >= interval) {

accumulator -= interval;

}

render();

if ( !window.isvSync() ) {

sync();

}

}

}

**Фиг. 3.2 gameLoop**

„timer” е инстанция на класа Timer. Функцията getElapsedTime() от класа Timer връща времето от миналото извикване на функцията.

При създаване на обект от типа GameEngine се създава обект от типа Window.

В този обект се сетъпва прозореца и се задават OpenGL променливи, така че OpenGL да прави Depth test, който е нужен за да не се случва по-далечни обекти да се рендерират пред по-близки предмети, и да се използва функция на OpenGL, чрез която само лицата, които са насочени към нас се рендерират, а останалите биват отхвърлени, което премахва приблизително 50% от лицата от екрана, което ускорява рендерирането.

При затваряне на прозореца се извиква функцията cleanup от GameEngine-a, която извиква функцията cleanup от игровата логиката, за да се погрижи за изчистване на памет, която е заела чрез OpenGL обекти (Mesh).

IGameLogic е интерфейс, за логиката на играта, по който трябва да се направи клас, за да се направи игра използвайки класа GameEngine.

**3.2.2 Шейдъри**

Шейдърите се използват в класа Renderer за да покажат 3D обектите на екрана. Шейдърите се зареждат чрез инстанция на класа ShaderProgram.

Класа ShaderProgram се занимава със зареждането на vertex и fragment шейдърите. Както всеки клас използващ OpenGL и в този имаме функция cleanup, която трябва да извикаме, като затворим приложението.

В нашият Renderer обект задаваме projectionMatrix-а на инстанцията на ShaderProgram. Projection матрицата се използва за да изглежда, че по-далечните обекти са по-малки, тъй като това нормално не се поддържа от OpenGL. Projection матрицата освен това контролира и FOV (Field of view) на героя, както и колко на близо могат да се рендерират обектите и до колко надалеко могат да се рендерират.

Една друга матрица, която ползваме, е View матрицата. Тази матрица се използва за да се премести и завърти всеки обект, така че да изглежда сякаш местим и въртим камерата. Този ефект се получава, като умножим матрицата по координатите на обекта.

В класа Renderer имаме функция render, на която и подаваме прозореца, камерата и обектите за рендериране. Тази функция е показана на фигура 3.3.

public void render(Window window, Camera camera, Mesh[] meshes) {

clear();

if ( window.isResized() ) {

glViewport(0, 0, window.getWidth(), window.getHeight());

window.setResized(false);

}

shaderProgram.bind();

// Update projection Matrix

Matrix4f projectionMatrix = transformation.getProjectionMatrix(FOV, window.getWidth(), window.getHeight(), Z\_NEAR, Z\_FAR);

shaderProgram.setUniform("projectionMatrix", projectionMatrix);

// Update view Matrix

Matrix4f viewMatrix = transformation.getViewMatrix(camera);

shaderProgram.setUniform("texture\_sampler", 0);

// Render each mesh

for(Mesh mesh : meshes) {

Matrix4f modelViewMatrix = transformation.getModelViewMatrix( viewMatrix);

shaderProgram.setUniform("modelViewMatrix", modelViewMatrix);

// Set model view matrix for this mesh

mesh.render();

// Render the mesh

}

shaderProgram.unbind();

}

**Фиг. 3.3 render**

Класа Mesh се използва, за да се създадат по-лесно OpenGL обектите, които ползваме за съхранение на Vertex-ите, Index-ите, текстурата и координатите на текстурата върху обекта. Функцията cleanup на обекта извиква изтриването на буферите, които се използват за съхраняване на тази информация.

Вертекс шейдъра е програмируема част от преубразуването на 3D сцена в 2D картина, която да се покаже на екрана. Вертекс шейдъра се използва, за да премести Vertex-ите на правилните места, за да създаде илюзията за разстояние и камера. Тук се подават матриците, които се използват за тази цел.

#version 330

layout (location=0) in vec3 position;

layout (location=1) in vec2 texCoord;

out vec2 outTexCoord;

uniform mat4 modelViewMatrix;

uniform mat4 projectionMatrix;

void main()

{

gl\_Position = projectionMatrix \* modelViewMatrix \* vec4(position, 1.0);

outTexCoord = texCoord;

}

**Фиг. 3.4 Vertex шейдър**

Fragment шейдърите се използват за оцветяването на дадени части от екрана, според зададени текстури и данните, които Vertex шейдъра е обработил (outTexCoord).

#version 330

in vec2 outTexCoord;

out vec4 fragColor;

uniform sampler2D texture\_sampler;

void main()

{

fragColor = texture(texture\_sampler, outTexCoord);

}

**Фиг. 3.5 Fragment шейдър**

**3.2.3 Игрова логика**

**3.2.3.1 Dummy Game**

Игровата логика (DummyGame) е написана като имплементация на интерфейса IGameLogic. При създаването на нов обект от тип DummyGame се създават нужните обекти за работа на играта (стакове за кои чънкове да се зареждат, премахват или презареждат и обекта Renderer)

Във **функцията init** се извиква функцията init на обекта Renderer, използвайки даденият ни прозорец(Window) и се сетъпва света (World).

Функцията input обработва входните данни на играча.

Във фигура 3.6 е показана частта от кода, която обработва input от позицията на мишката на екрана и я поставя в средата на екрана.

double[] mouse=window.pullMouseData();

//reads the mouse position

window.setMouseData(window.getWidth()/2, window.getHeight()/2);

//sets the mouse position to the center of the screen

Vector3f rotation=World.player.getRotation();

//get the camera rotation

rotation.x+=(float) mouse[1];

rotation.y+=(float) mouse[0];

//adds the mouse position to the rotation angles

if(rotation.x>90){

rotation.x=90;

}

if(rotation.x<-90){

rotation.x=-90;

}

//makes the rotation along the x axis locked between 90° and -90°

if(rotation.y<-180){

rotation.y+=360;

}

if(rotation.y>180){

rotation.y-=360;

}

//makes so the y rotation is within the bounds of 180° and -180°

World.player.setRotation(rotation.x, rotation.y, rotation.z);

//set the rotation of the camera

**Фиг. 3.6 Обработване на позицията на мишката**

След като се обработи позицията на мишката, се обработва входа от клавиатурата. Използва се функцията isKeyPressed от прозореца и според това се променя булева променлива, за да се използва това дали даден бутон е натиснат, без да трябва да се извиква функцията отново.

if(keyW^window.isKeyPressed(GLFW\_KEY\_W)){

keyW= !keyW;

}

// Checks the state of key W and if it has changed, we change the variable keyW

if(keyS^window.isKeyPressed(GLFW\_KEY\_S)){

keyS= !keyS;

}

// Checks the state of key S and if it has changed, we change the variable keyS

if(keyA^window.isKeyPressed(GLFW\_KEY\_A)){

keyA= !keyA;

}

// Checks the state of key A and if it has changed, we change the variable keyA

if(keyD^window.isKeyPressed(GLFW\_KEY\_D)){

keyD=! keyD;

}

// Checks the state of key D and if it has changed, we change the variable keyD

if(key1^window.isKeyPressed(GLFW\_KEY\_1)){

key1= !key1;

if(key1){

type=0;

}

}

// Checks the state of key 1 and if it has changed, we change the variable key1

if(key2^window.isKeyPressed(GLFW\_KEY\_2)){

key2= !key2;

if(key2){

type=1;

}

}

// Checks the state of key 1 and if it has changed, we change the variable key2

**Фиг. 3.7 Обработване на вход от клавиатурата**

Като сме обработили входа от клавиатурата и позицията на мишката остава само да обработим бутоните от мишката.

if (mouseButton1 ^ window.isMouseKeyPressed(GLFW\_MOUSE\_BUTTON\_1)){

//tests for a change in mouse button 1’s state

mouseButton1= !mouseButton1;

//sets mouseButton1 to true or false, according to the button’s state

if(mouseButton1){

//has the button been pressed or unpressed

Thread dig;

//creating a thread for digging

dig = new Thread(){

@Override

public void run(){

Vector3f pos=World.player.getPosition();

Vector3f rot=World.player.getRotation();

//gets the player’s position and rotation

Vector3d rotation=Math3d.rotateY(Math3d.rotateX(new Vector3d(0,0,1),rot.x), -(rot.y+180));

//creating a vector pointing in the direction that the player is watching at

Vector<Point3d> cubes=Math3d.LineIntersections(new Point3d(pos.x,pos.y,pos.z), rotation, 5);

//checking what cubes are being intersected by the view vector

compare=new Point3d(pos.x,pos.y,pos.z);

cubes.sort(new Comparator(){

@Override

public int compare(Object o1, Object o2) {

return (int)((Math3d.distance(compare, (Point3d)o1) -

Math3d.distance(compare, (Point3d)o2))\*10000);

}

});

//sorting the points according to how far away they are from the player

Point3d intersection=null;

rotation.multiply(5);

//multiplies the view vector’s length by 5 (it was previously 1)

for(int i=0;(i<cubes.size())&&(intersection==null);i++){

intersection=World.intersects(cubes.get(i),new Line3d(new Point3d(pos.x,pos.y,pos.z),

new Point3d(pos.x+(rotation.x),pos.y+(rotation.y),pos.z+(rotation.z))));

}

//checks for an intersection between the view vector and

//the objects inside of the cubes that intersect the view vector

if(intersection==null) return;

World.dig(intersection, 1);

//if there is an intersection point, remove a part of the objects at the given place

}

};

dig.start();

//start the dig thread

}

}

**Фиг. 3.8 Код, който се изпълнява при натискане на левият бутон на мишката**

След като се провери дали бутонът е натиснат и не е бил натиснат при миналото извикване на input функцията, се създава нов Thread, в който според местоположението на играча и ориентацията му се премахва част от терена.

За добавяне към терена се проверява за десният бутон на мишката и се използва нишка, подобна на нишката за премахване от терена, но вместо извикване на функцията dig от класа World се извиква функцията place, с параметри „intersection”, „1.0d” и „type”. „ type” е променлива, според която се добавя определен вид обект към терена (например трева или камък).

Освен обработване на входните данни, в DummyGame класа се ъпдейтва играта и се рендерира.

**Функцията update** като бъде извикана и е зададено колко време е минало от миналото и извикване, за да се знае колко големи да са промените от ъпдейтването.

Във фигура 3.9 е показано как камерата се мести според времето минало от предишното извикване на функцията, за да се движи камерата с еднаква скорост дори тази функция да бъде извикана след много или малко време.

if(keyW){

World.player.movePosition(0, 0, -1.5f\*interval);

}

//move camera forward

if(keyA){

World.player.movePosition(-1.5f\*interval, 0, 0);

}

//move camera left

if(keyS){

World.player.movePosition(0, 0, 1.5f\*interval);

}

//move camera back

if(keyD){

World.player.movePosition(1.5f\*interval, 0, 0);

}

//move camera right

**Фиг. 3.9 местене на камерата**

В update функцията освен местене на камерата се зареждат нови парчета от света, премахват се парчета, които са твърде надалеко и се пресъздава 3D обекта на парчетата, които са били променени.

if(checking<=0){

//checks weather it has been more than half a second since the last checking thread has ended

checking=1000;

//sets the value of checking to a big enough value to not run two checking threads at once

checkingThread=new Thread(){

@Override

public void run(){

Vector3f position =World.player.getPosition();

int x=(int)Math.floor(position.x/5);

int y=(int)Math.floor(position.y/5);

int z=(int)Math.floor(position.z/5);

//turn the position of the player to the integer locations for the chunk the player is in

for(int i=x-3;i<=x+3;i++){

for(int j=y-3;j<=y+3;j++){

for(int l=z-3;l<=z+3;l++){

if(i==x-3||i==x+3||j==y-3||j==y+3||l==z-3||l==z+3){

if(World.isChunkLoaded(new Point3d(i,j,l))){

if(!discard.contains("("+i+";"+j+";"+l+")")){

discard.add("("+i+";"+j+";"+l+")");

discardp.add(new Point3d(i,j,l));

}

}

//if a chunk should unload and isn’t in the Vectors for unloading, add it

}else{

if(!World.isChunkLoaded(new Point3d(i,j,l))){

if(!load.contains("("+i+";"+j+";"+l+")")){

World.CreateChunk(new Point3d(i,j,l));

load.add("("+i+";"+j+";"+l+")");

loadp.add(new Point3d(i,j,l));

}

//if a chunk should load and isn’t in the vectors for loading, add it

}else{

if(World.getchunk(new Point3d(i,j,l)).isChanged()){

if(!redo.contains("("+i+";"+j+";"+l+")")){

World.getchunk(new Point3d(i,j,l)).readyMesh();

redo.add("("+i+";"+j+";"+l+")");

redop.add(new Point3d(i,j,l));

}

}

//if a chunk has been changed and isn’t in the vectors for reloading, add it

}

}

}

}

}

checking=0.5f;

//set the checking variable to 0.5

}

};

checkingThread.start();

} //remove from the variable checking (in seconds) the amount of time(in seconds)

checking-=interval; //since the last call of the function update

**Фиг. 3.10 зареждане, премахване и презареждане на парчета от терена.**

Обработката на обектите и подготвянето им да се добавят като OpenGl обекти се извършва в нишка, отделна от главната, за да не се забавя главната нишка да извършва допълнително изчисления. Поради тази причина нови парчета терен първо са създадени в нишката checkingThread, където може да се изпълнят по-бавните изчисления, а после извън тази нишка да се сглобят OpenGl обектите. Това се случва в кода от фигура 3.11, където се казва, че ако това извикване на функцията update до тук е изминало за по-малко милисекунди от миналото, се зарежда едно парче терен, презарежда се едно и се премахва едно стига да има извикване за зареждане/презареждане/ премахване.

if(((System.currentTimeMillis()-time)<(interval\*1000))

&&((!discard.isEmpty())||(!load.isEmpty())||(!redo.isEmpty()))){

if(!discardp.isEmpty()){

System.out.println(discardp.size());

int size=discardp.size();

try {

World.DiscardChunk(discardp.get(size-1));

} catch (Exception ex) {

Logger.getLogger(DummyGame.class.getName()).log(Level.SEVERE, null, ex);

}

discardp.remove(size-1);

discard.remove(size-1);

}

if(!loadp.isEmpty()){

int size=loadp.size();

try {

World.generateMesh(loadp.get(size-1));

} catch (Exception ex) {

Logger.getLogger(DummyGame.class.getName()).log(Level.SEVERE, null, ex);

}

loadp.remove(size-1);

load.remove(size-1);

}

if(!redop.isEmpty()){

int size=redop.size();

try{

World.redoChunk(redop.get(size-1));

} catch (Exception ex) {

Logger.getLogger(DummyGame.class.getName()).log(Level.SEVERE, null, ex);

}

redo.remove(size-1);

redop.remove(size-1);

}

}

**Фиг. 3.11 създаване на нови OpenGl обекти и премахване на стари**

**Функцията render** на класа DummyGame извиква функцията render на инстанция на класа Renderer, подавайки и прозореца (window), в който да рисува, камерата, и 3D OpenGl обектите, които да се покажат на екрана.

Тези обекти се взимат от класа World.

**cleanup функцията**, показана на фигура 3.12, извиква cleanup методите на инстанцията на класа Renderer и на класа World.

@Override

public void render(Window window) {

renderer.render(window,World.player, World.GetMeshes());

}

@Override

public void cleanup() {

renderer.cleanup();

World.cleanup();

}

**Фиг. 3.12 render и cleanup**

**3.2.3.2 World**

**Класа World** е финален клас, в който всички методи и променливи са статични и конструктора е скрит. По този начин симулирам клас с една единствена инстанция, която може да се достъпи от всякъде.

За да се използва класа трябва първо да се извика функцията setup, която създава HashMap, в който да се пазят парчетата терен и камерата, действаща като играча.

**Функцията DiscardChunk** приема за стойност точка в 3D пространството и ако на тази точка отговаря парче терен, това парче терен бива премахнато. За тази цел се извиква функцията cleanup на даденото парче и след това се премахва от HashMap-а.

**Функцията redoChunk** пресъздава OpenGl обекта на парчето терен за да се покажат промените на екрана. Това се случва като първо се изчисти старият OpenGl обект, като се извика неговата функция cleanup, а след това се създава нов 3D OpenGl обект, в който да се съдържат новите лица на обекта.

Обектът бива изтрит и пресъздаден, защото като се работи на ниско ниво не е лесно да се създаде масив, който да работи като списък, а дори да може, една такава имплементация на списък, с добавяне и премахване на обекти (вертекси и лица) ще е много по-бавно от този начин на работа.

**Функцията CreateChunk** създава ново парче земя, което се добавя към HashMap-а с ключ string, който е създаден от координатите на даденото парче земя по начина, показан във фигура 3.13.

Координатите на парчето терен биват превърнати в int , защото парчетата терен биват поставени на цели координати и, ако се използват самите координати за ключа, е възможно малките несигурности в стойността на променливата double x да доведат до различен ключ.

"("+(int)x+";"+(int)y+";"+(int)z+")"

**Фиг. 3.13 ключ за парче терен с координати x, y и z**

**Функцията isChunkLoaded** връща резултат true или false според това дали съществува ключ създаден от подадените координати в HashMap-а.

**GetMeshes** връща всички обекти за рендериране, които да се подадат на инстанция на класа Renderer. Това се случва като се обходят всички ключове от HashMap-а и за всеки ключ, за който отговаря парче земя със съществуващ OpenGl обект, се добавя този OpenGl обект към Set, който е преубразуван в масив. Ако в момента няма заредени парчета земя, които имат създаден OpenGl обект, се връща празен масив.

**Функцията cleanup** извиква функцията cleanup на всички парчета земя, за да се премахнат OpenGl обектите.

**getChunk** приема като аргумент точка в 3D пространството и връща координатите на парче терен, в чиито граници е тази точка.

**IsLoaded** приема като аргумент точка в 3D пространството и връща като стойност true или false според това дали парчето земя, в чиито граници е тази точка , е заредено.

**Функцията IsAir** връща стойност true или false според обема, който обектите в дадено кубче заемат - true ако обемът е по-малко от 1Е-10 или false ако обемът е повече от тази стойност.

**Функцията readyMesh** приема като вход позиция на парче земя и ако е заредено му създава OpenGl обект, който да се използва за рисуването на екрана. Тази функция може да се изпълнява само в главната нишка на играта, защото за да се използва OpenGl трябва да се извикат някои функции, но тези функции дават позволение само на една нишка да използва OpenGl. Ако друга нишка иска и тя да използва OpenGl трябва и тя да извика тези функции, но те не са бързи и не може да се прехвърлят OpenGl обекти от една нишка на друга. Поради тези причини тази функция само сглобява обекта, който да бъде изведен на екрана. По-голямата част от действието се случва в друга функция-readyMesh. Тази функция индиректно създава и подрежда всички нужни данни за да се създаде обект за рендериране, като извиква функция със същото име от парчето терен за което трябва да се случи подготовката.

**intersects** приема за стойност координата на кубче в света и 3D линия и връща като стойност къде те се пресичат. Това се случва, като бъде извикана функция със същото име от парчето терен, към което принадлежи това кубче.

**Функцията hasAir** проверява дали обемът на обект в дадено кубче надвишава 1 - 1Е-10. Ако обемът е повече, функцията връща false.

**getchunk** взима като стойност координати на парче терен и връща като стойност това парче терен.

**Функцията dig** се използва за премахване на обем от терена. Това се случва, като се извика функцията dig за всяко парче терен за всяко кубче, което е засегнато от промяната. Тя приема за вход точката на промяната и диаметъра.

Тези параметри биват подадени нататък към парчето терен.

**Функцията place** се използва за добавяне на обем от терена. Това се случва, като се извика функцията place за всяко парче терен за всяко кубче, което е засегнато от промяната. Освен точка на промяна и диаметър, тази функция освен това приема за стойност какъв тип обем да се добави към земята (пръст или камък)

**3.2.3.3 chunk**

**chunk** (парче терен) е клас, в който се запазват кубчета по същият начин, по който се запазват парчета терен в класа World. Парчетата терен съдържат по 125 кубчета.

Този клас има cleanup метод, който премахва OpenGl обекта, асоцииран с дадената инстанция на обекта, като преди това му извиква метода cleaup.

**Функцията New** създава всички кубчета, принадлежащи на това парче терен.

**readyMesh** е функция, която подготвя обекта за създаване на OpenGl обекта на парчето терен (Фиг. 3.14). Координатите на точките и съответните им координати от текстурата са запазени във Float вектори.

Vector<Float> positions;

Vector<Float> texCoords;

public void readyMesh(){

String[] keys=cubes.keySet().toArray(new String[cubes.size()]);

//getting the keys from the HashMap

positions=new Vector();

texCoords=new Vector();

Cube c;

for (String key : keys) {

//iterating over every key from the HashMap

c = cubes.get(key);

if(c.isChanged())c.setupVectors();

// if the cube has been changed, remake its vectors

positions.addAll(c.getPoints());

// adds the vector of positions from the current cube to the chunk

texCoords.addAll(c.getTexture());

// adds the vector of texture coordinates from the cube to the chunk

}

changed=false;

//set the changed flag to false

}

**Фиг. 3.14 подготовка на обекта за създаване на обект от типа Mesh**

**Функцията createMesh** създава обект от типа Mesh за даденото парче терен.

Обекта Mesh е създаден за да се справя с OpenGl буферите, които са нужни за да можем да нарисуваме даден 3D обект на екрана.

**Методът getMesh** връща обекта от тип Mesh, който принадлежи на това парче терен.

**Методът hasMesh** проверява дали даденото парче терен има Mesh, който да се покаже на екрана.

**Методът discardMesh** работи идентично на cleanup метода, но cleanup метода се извиква само при затваряне на програмата или премахване на парчето терен, докато метода discardMesh се ползва само за временно премахване на обекта Mesh

isAir и hasAir са препратки към същите функции от кубчетата.

**Функцията intersects** проверява дали дадено кубче пресича 3D линия. Ако я пресича се връща точката на пресичане, ако не се пресича - се връща null.

**Функцията dig** приема координати на кубче, координати на промяната и диаметър на промяната. Тя премахва от кубчето част от обекта за проверка за докосване и част от обекта, който се показва на екрана.

**Функцията place** приема координати на кубче, координати на промяната, диаметър на промяната и типа обект, който да се добави (трева или камък).

Тя добавя към кубчето към обекта за проверка за докосване и към обекта, който се показва на екрана.

Освен промяната на кубчетата, функциите dig и place вдигат флага changed на парчето терен за да му бъде пресъздаден обекта за рендериране.

**setChanged** вдига или смъква флага changed.

**isChanged** връща като стойност стойността на флага changed.

**3.2.3.4 Cube**

**Класа Cube** е парче от терена с размер 1 на 1 на 1.

В едно кубче се запазват обекти от типа Solid.

Solid e клас от библиотеката J3DBool.

Тази библиотека се занимава с променяне на 3D обекти използвайки булеви оператори. Такива операции са: обединяване, разлика и пресичане.

**Функцията New** създава кубче с определен тип обекти според височината на кубчето. Ако е между 0 и -3 кубчето е трева. Ако е под -3 кубчето е камък, а ако е над 0, кубчето е въздух (няма нищо в него) *(Фиг. 3.15)*

public void New(Point3d loc){

if(loc.y>0){

//the cube is air

}else if(loc.y<=0 && loc.y>-3){

bound=Math3d.getCubeAt(loc, 1);

Materials.put(grass, Math3d.getCubeAt(loc, 1));

//the cube is grass

}else if(loc.y<=-3){

bound=Math3d.getCubeAt(loc, 1);

Materials.put(stone, Math3d.getCubeAt(loc, 1));

//the cube is stone

}

location=loc;

}

**Фиг. 3.15 Създаване на нов куб.**

**removeBound** премахва от обекта, използван за проверка на обема на кубчето и проверки за пресичане с 3D линия.

**removeMaterial** *(Фиг.3.16)* премахва от обектите, които се показват на екрана.

public void removeMaterials(Solid s){

Integer[] materials=Materials.keySet().toArray(new Integer[Materials.size()]);

for (Integer material : materials) {

Materials.put(material, new BooleanModeller(Materials.get(material), s).getDifference());

//remove the Solid s from every material in the HashMap materials

}

parent.setChanged(true);

//set the flag changed of the parent chunk to true

}

**Фиг. 3.16 премахване от обектите, които се показват на екрана**

**addBound** добавя към обекта използван за проверка на обема на кубчето и проверки за пресичане с 3D линия.

**addMaterial** *(Фиг. 3.17)* добавя към обектите, които ще се покажат на екрана.

public void addMaterial(Solid s,Integer m){

Solid mat=new Solid();

if(Materials.containsKey(m)){

//does the cube contain the key m already

mat=Materials.get(m);

if(mat.isEmpty()){

//is the Solid in the HashMap under that key empty

mat=new BooleanModeller(s,Math3d.getCubeAt(location, 1)).getIntersection();

//if it is empty just set the key in the HashMap to the part of the Solid s that intersects the cube

}else{

Solid s1=new BooleanModeller(s, Math3d.getCubeAt(location, 1)).getIntersection();

if(!s1.isEmpty()){

mat=new BooleanModeller(s1,mat).getUnion();

}

//if it is not empty add the part of it that is inside of the cube to the Solid under the key m

}

}else{

mat=new BooleanModeller(s,Math3d.getCubeAt(location, 1)).getIntersection();

System.out.println(mat.isEmpty());

If the key isn’t in put it in with the part of the Solid s that is inside the cube

}

parent.setChanged(true);

Materials.put(m, mat);

}

**Фиг. 3.17 добавяне на материал за показване на екрана**

**Функцията crosses** *(Фиг. 3.18)*проверява дали дадена отсечка пресича 3D обекта (Solid) в дадено кубче. Това се случва като се проверява всяко триъгълно лице на обекта и се проверява дали линията пресича някое лице.

public boolean Crosses(Line3d a){

Point3d[] points=bound.getVertices();

int[] indices=bound.getIndices();

for(int i=0;i<(indices.length/3);i++){

Point3d p1=points[indices[i\*3]];

Point3d p2=points[indices[i\*3+1]];

Point3d p3=points[indices[i\*3+2]];

//p1, p2 and p3 are 3 points that create a triangle face

if(equals(volumeOfTrianglePyramid(a.a, p1,p2,p3),

(volumeOfTrianglePyramid(a.a, a.b,p1,p2)+volumeOfTrianglePyramid(a.a,a.b,p2,p3)

+volumeOfTrianglePyramid(a.a, a.b,p1,p3))-volumeOfTrianglePyramid(a.b, p1,p2,p3))){

return true;

//returns true if the line crosses a face

}

}

return false;

//returns false if the line didn’t cross any one of the faces

}

**Фиг. 3.18 проверка за пресичане**

|  |  |
| --- | --- |
| Във фигура 3.19 имаме пример за триъгълно лице (ABC) и линия, която го пресича (DE). Проверката дали една линия пресича дадено лице се случва по следният начин:  1.Взима се обема на АBDE и се добавя към обема на ACDE и обема на BCDE.  2. От полученият обем изваждаме обема на пирамидата АBCE  3. Сравняваме полученият обем с обема на пирамидата ABCD, ако получените обеми са равни, това означава че DE пресича ABC | **Фиг. 3.19 пример за проверката.** |

**-equals** е функция, която може да се достъпи само от този обект и сравнява дали 2 числа от тип double са в допустими граници, да кажем, че са приблизително равни (това означава да са в границите ±1Е-10 едно от друго).

**Функцията setupVectors** създава вектор Points и вектор Texture, които после могат да се вземат с функциите getPoints и getTexture.

Тези вектори се ползват от парчето терен към което принадлежи това кубче за да се покаже на екрана.

Векторите ще са празни, ако кубчето е празно или обектите във всички

6 кубчета около даденото кубче имат обем 1 и обектът в това кубче

също има обем 1.

**isAir** проверява дали в кубчето обектът няма лица.

**hasAir** проверява дали обемът на обекта в кубчето е по-малък от 1- 1Е-10

**Функцията getCrossPoint** връща точката на пресичане между обекта в кубчето и линия.

**setChanged** вдига или смъква флага changed на кубчето.

**getChanged** връща стойността на флага changed на кубчето.

**3.2.3.5 Math3D**

**Math3D** е клас, в който се намират функции, свързани с 3D математикa.

**Функцията lineSolidIntersection** *(Фиг.3.20)* връща точка, където обект от типа Solid и линия се пресичат.

public static Point3d lineSolidIntersection(Solid s,Line3d l){

Point3d closest=null;

double distance=Double.MAX\_VALUE;

Point3d[] points=s.getVertices();

int[] indices=s.getIndices();

for(int i=0;i<(indices.length/3);i++){

Point3d p1=points[indices[i\*3]];

Point3d p2=points[indices[i\*3+1]];

Point3d p3=points[indices[i\*3+2]];

if(equals(volumeOfTrianglePyramid(l.a, p1,p2,p3),Math.abs(

volumeOfTrianglePyramid(l.a, l.b,p1,p2)+volumeOfTrianglePyramid(l.a,l.b,p2,p3)+

volumeOfTrianglePyramid(l.a, l.b,p1,p3)-volumeOfTrianglePyramid(l.b, p1,p2,p3)))){

Point3d p=linePlaneIntersectionPoint(p1, p2, p3, l);

if(p!=null &&distance(p, l.a)<distance){

distance=distance(p,l.a);

closest=p;

}

}

}

return closest;

}

**Фиг. 3.20 намиране на пресечна точка между обект и линия**

**linePlaneIntersection** намира пресечната точка между линия и равнина описана чрез 3 точки.

**Функцията volumeOfTrianglePyramid** връща обема на триъгълна пирамида.

Обемът е пресметнат, пресмятайки детерминанта на матрица 4 на 4 и разделяйки я на 6. В матрицата първият ред е x координатите на точките, вторият ред е y координатите на точките, третият ред е z координатите на точките, а четвъртият ред е 4 единици.

**Функциите rotateX, rotateY и rotateZ** завъртат вектор съответно

около X, Y или Z оста.

**distance** връща разстоянието между две точки в пространството.

**getCubeAt** приема аргументи точка, където да се сложи центъра на куба и дължина на страната на куба. Функцията зарежда обект от типа Solid, увеличава/намалява го до нужният размер и го мести в пространството.

**getClosest** приема за стойност вектор с точки и една точка, с която да сравнява. Функцията връща точката от вектора, която се намира най-близо в пространството до точката за сравняване.

**Функцията convertArrayFloat** преобразува масив от елементи тип Float до масив с елементи от тип float.

**Функцията LineIntersections** приема като аргументи точка , вектор и дължина и връща като резултат през кои кубчета, с ръб едно и център разположен на точка с координати цели числа , преминава този вектор.

За улеснение тази функция извиква функцията lineIntersections2D, която се извиква с xy координатите на вектора, xz координатите на вектора и yz координатите на вектора. Ако векторът пресича едно кубче, той ще пресича и съответните квадрати в тези 2D представяния на проблема.

По този начин, ако и в трите 2D представянията на проблема вектора пресича дадено кубче, това означава че 3D вектора пресича това кубче.

**LineIntersections2d** приема като стойност едно от 2D представянията на проблема и връща двуизмерен масив с булеви променливи, показващи кои квадрати в 2D представянето пресича вектора.

**LineIntersects** проверява дали определено квадратче в 2D репрезентацията на проблема е пресечено от вектора.

**f e функция**, която приема за стойност a, b и x и връща a\*x+b

**Четвърта глава**

**Ръководство на потребителя**

За да се стартира играта на даден компютър е необходимо да има инсталация на Java поне версия 8.

Играта може да се изтегли от <https://github.com/pepi-tsankov/game> , където трябва да се натисне на бутона „Clone or download” и след това „DownloadZIP”.

След като се изтегли архива, той трябва да се разархивира.

Играта може да се стартира от файла GameTest-1.0,jar.

След като се стартира играта, потребителят може да се оглежда с мишката.

Потребителят може да се мести използвайки бутоните WASD.

Играчът може да промени терена с ляв и десен бутон на мишката,

като левият бутон премахва част от терена, а десният добавя към терена.

За да се промени какво се поставя чрез десният бутон на мишката се използват бутоните 1 и 2 от клавиатурата.

Когато потребителят иска да затвори приложението, трябва да натисне бутона Esc от клавиатурата.

**Заключение**

**В дипломната работа са постигнати следните цели:**

-Играта успешно създава терен

-Играча може да се предвижва през терена

-Терена може да бъде модифициран по време на играта

-Играта може да поддържа 2 или повече типа обекти, които да са част от терена

**Идеи за бъдещо развитие:**

-Използване на 2D и 3D noise функция за създаване на по-реалистичен терен и пещери.

-Променяне на начина, по който се слагат текстурите на терена за да изглежда по-добре

-Добавяне на възможността на играча да минава върху терена и да може да скача а не да пътува на една и съща височина.

-Разработка на multiplayer версия на играта

**Използвана литература**

**1. Oracle Help Center** [**https://docs.oracle.com/javase/8**](https://docs.oracle.com/javase/8)

**2. StackOverflow** [**http://stackoverflow.com/**](http://stackoverflow.com/)

**3. LightHouse3d** [**http://www.lighthouse3d.com/tutorials/maths/ray-triangle-intersection/**](http://www.lighthouse3d.com/tutorials/maths/ray-triangle-intersection/)

**4. had2know** [**http://www.had2know.com/academics/tetrahedron-volume-4-vertices.html**](http://www.had2know.com/academics/tetrahedron-volume-4-vertices.html)

**5. lwjgl** [**http://wiki.lwjgl.org/**](http://wiki.lwjgl.org/)

**6. gitbooks** [**https://www.gitbook.com/book/lwjglgamedev/3d-game-development-with-lwjgl/details**](https://www.gitbook.com/book/lwjglgamedev/3d-game-development-with-lwjgl/details)

**Съдържание**

|  |  |  |
| --- | --- | --- |
|  |  | **Стр.** |
|  | Увод | **4** |
| **I глава:** | Преглед на подобни игри и средства за създаването им | **5** |
| **1.1.** | Преглед на подобни игри | **5** |
| **1. 1.1** | Space Engineers | **5** |
| **1.1.2** | FortressCraft Evolved | **6** |
| **1.1.3** | Minecraft | **8** |
| **1.2** | Преглед на известните средства за създаване на игри. | **9** |
| **1.2.1** | Unreal engine 4 | **9** |
| **1.2.2** | OpenGL | **10** |
| **II**  **глава:** | Изисквания, средства за създаване на играта и използвани алгоритми | **11** |
| **2.1** | Изисквания към играта | **11** |
| **2.2** | Избор на средства за създаване на играта | **11** |
| **2.3** | Описание на алгоритъм | **12** |
| **III. глава** | Реализация на триизмерната игра | **13** |
| **3.1** | Основни понятия в игрите. | **13** |
| **3.2** | Описване на начина на реализация | **15** |
| **3.2.1** | Енджина | **15** |
| **3.2.2** | Шейдъри | **17** |
| **3.2.3** | Игрова логика | **20** |
| **3.2.3.1** | Dummy Game | **20** |
| 3.2.3.2 | World | **26** |
| 3.2.3.3 | Chunk | **30** |
| 3.2.3.4 | Cube | **33** |
| 3.2.3.5 | Math3D | **37** |
| **IV. глава** | Ръководство на потребителя | **41** |
|  | Заключение | **42** |
|  | Използвана литература | **43** |