TALLINN UNIVERSITY OF TECHNOLOGY
Faculty of Information Technology
Department of Computer Science
Chair of Network Software

GAME ENGINE FOR FOUR-DIMENSIONAL FIRST-PERSON SHOOTER

BACHELOR'S THESIS
ITV40LT

Student:	Jevgeni Krutov
Student code:	103923IAPB
Supervisor:	Jaagup Irve

Tallinn
2014

Autorideklaratsioon

Kinnitan, et olen koostanud antud lõputöö iseseisvalt ning seda ei ole kellegi teise poolt varem kaitsmisele esitatud. Kõik töö koostamisel kasutatud teiste autorite tööd, olulised seisukohad, kirjandusallikatest ja mujalt pärinevad andmed on töös viidatud.

(*kuupäev*) (*allkiri*)

Annotatsioon

Selle töö eesmärk on neljamõõtmelise tulistamismängu mootori loomine Java programmeerimiskeeles. Lisaeesmärgiks on veebipõhise mängutaseme redaktori loomine mängumootori demonstreerimise hõlbustamiseks.

Põhilised probleemid millega tegelesin selles töös on neljamõõtmelise objektide representeerimine andmestuktuuridena ja nende visualiseerimine ekraanil. Lisaks sellele tegelesin kõikide visualiseerimise etapide teostamisega ning mängu seisundi illustreeritavate indikaatorite, lihtsate füüsikaalgoritmide ja mängureeglite loomisega.

Selle töö tulemusena on arvutimäng, kus mängija võib liikuda neljadimensioonilises ruumis, hüpata, vastasi tulistada, asju koguda ja luua lihtsaid mängutasemeid.

Lõputöö on kirjutatud inglise keeles ning sisaldab teksti 41 leheküljel, 13 peatükki, 12 joonist.

Abstract

The aim of this work is to create a four-dimensional engine for first-person shooter game in Java programming language. In addition, a simple web-based level-redactor should be created for engine demonstration purposes.

The main problems I dealt with in this work is representation of four-dimensional objects as data structures and visualising them on screen. In addition to that, I dealt with implementing a full rendering pipeline, head-on display which provides feedback about the game state to the player, simple physics algorithms and game rools.

The result of this work is a game where player can move in four dimensions, jump, shoot enemies, collect items and create simple custom levels.

The thesis is in English and contains 41 pages of text, 13 chapters, 12 figures.

Glossary

OpenGL	***Open Graphics Library*** A cross-language, multi-platform application programming interface for rendering vector graphics [1]
LWJGL	***Lightweight Java Game Library*** An open source library for game development on Java platform [2]
GLSL	***OpenGL Shading Language*** A shading language, in which shaders are written [3]
4D	***Four-dimensional*** Having four spatial dimensions
Vertex	A point in space
Shader	A computer program that is usually executed on graphics processing unit of the graphics card [4]
Vertex shader	A shader that is executed for each vertex [5]
Fragment shader	A shader that computes color and other attributes of each fragment. The simplest kinds of pixel shaders output one screen pixel as a color value [5]
Rendering pipeline	A process consisting of sequential steps which results in two-dimensional representation of a scene on screen [6]
X, Y, Z, W	The names of the four coordinate axes of four-dimensional space
Ana / kata	The names of the two directions in which a four-dimensional object can move along W-axis [7]

Table of Contents

1. Introduction

Computer game production is a rapidly growing industry with billions of dollars of the annual worldwide revenue [8]. A significant amount of computer games of different genres has been made and the player's gaming experience improves as the industry develops.

1.1 Background and Problem

A computer game can be programmed by a single person. Some of the well-known best-selling games are made by such individuals, for example "Minecraft" [9] or "Papers, Please" [10].

This work is my attempt to create a game prototype with nontrivial gaming process that can be developed further and serve as the basis of a finished game product in the future.

The genre of the game is chosen to be first-person shooter as it is most demonstrative in sense of exploring four-dimensional world and can provide immersive real-time experience for the player.

1.2 Main Goals

The goal of this work is to create a basic game engine where player can observe four-dimensional space, jump, shoot, move and perform other simple actions in it.

A simple physical interaction algorithms of objects should be implemented.

As an addition, a simple level-redactor should be created for demonstration of four-dimensional level construction process.

1.3 Technique

The main tools to be used for engine creation are Java programming language and LWJGL library. Shaders are written using GLSL language.

The level-redactor is web-based (HTML, CSS, Javasript, PHP, JQuery).

1.4 Overview

The game source code is located at the address https://code.google.com/p/4d-game/

When the game launches, the menu with a list of levels is shown to the player. When selection is performed, the game starts (Figure 1). The player can move in three dimensions of the 4D-world using W, A, S, D keys, jump using space bar, shoot enemies using left mouse button. Q and E buttons are used to observe world in two different directions of a fourth spatial dimension. If the right mouse button is pressed at the moment of observing fourth dimension in some direction, the transition of player in fourth spatial coordinate is performed to the direction observed.

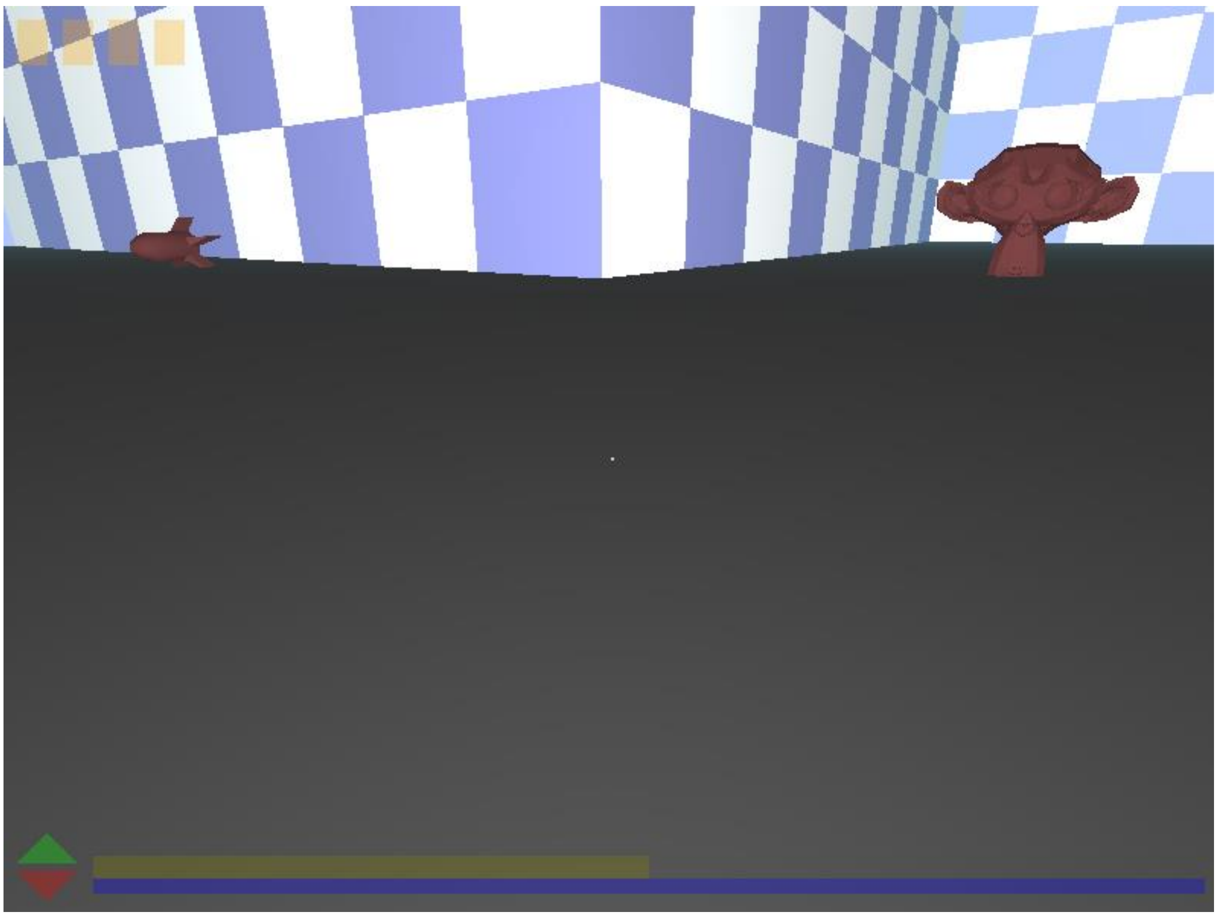


Figure 1 – screenshot of the game: ground, walls, pickable rocket item, enemy, head-up display

When observation of the world along one of the two directions of a fourth dimension is performed, the color filter is applied to the screen to indicate the possibility of transition to the direction observed (Figure 2). If the player is allowed to go in one direction, the green filter is applied; another direction – red filter. Black and white color filter is applied when the player is not allowed to go to the observed direction due to the probable collision with objects, that are located at the place of the desired transition.

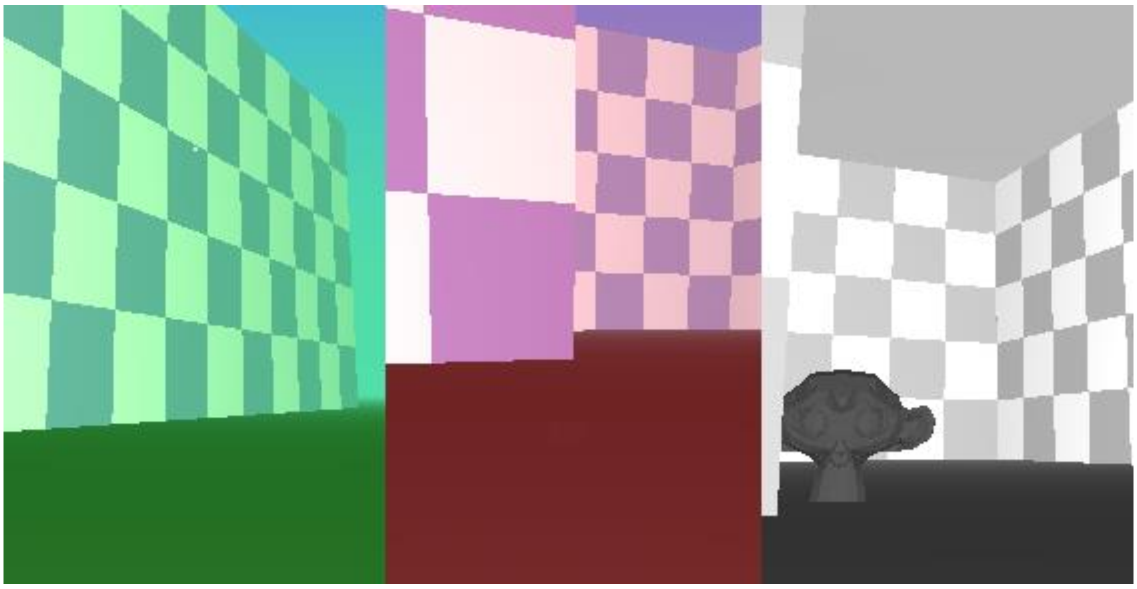


Figure 2 – color filters

The player can pick items (Figure 3) such as medkit that restores health (health is not regenerated), W-energy that quickly restores energy needed to perform W-transition (W-energy slowly regenerates) and additional rockets. Exit is an item as well.

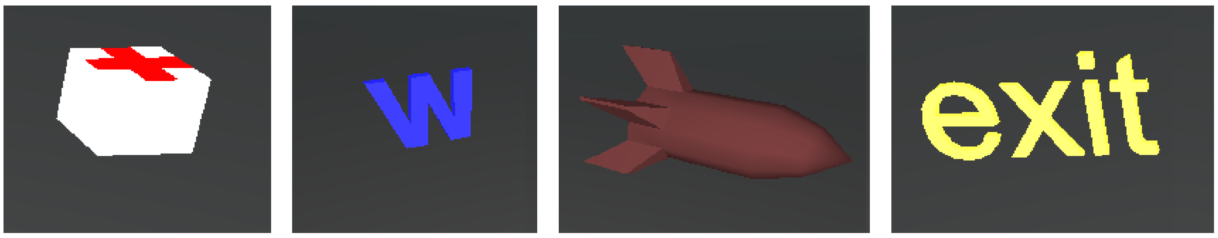


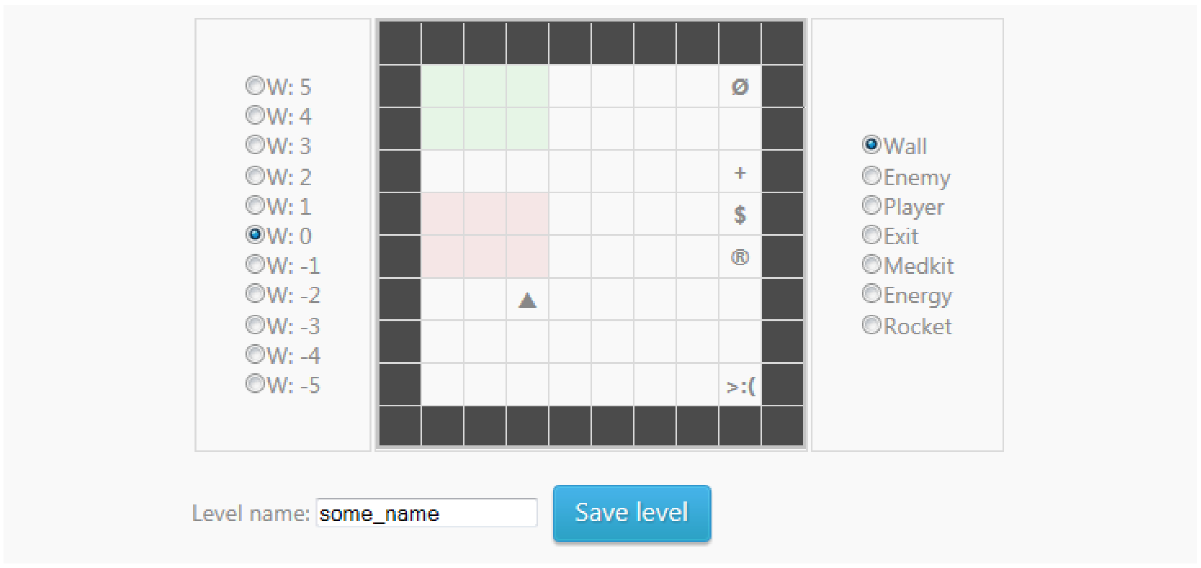
Figure 3 – items: medkit, W-energy, rocket, exit

The goal of the game is to find exit from the level and not be killed by the enemies.

1.5 Level-Redactor Overview

The level-redactor is located at the address http://dijkstra.cs.ttu.ee/~t103923/levelRedactor/

In level-redactor (Figure 4) it is possible to draw the map of a 4D level. There is a current fourth coordinate selector, an editable level map and a current editing mode selector. After entering the level's name and saving it, this level can be selected in game's level list.


W: 5
W: 4
W: 3
W: 2
W: 1
[✓ W: 0]
W: -1
W: -2
W: -3
W: -4
W: -5

[✓ Wall]
Enemy
Player
Exit
Medkit
Energy
Rocket

Level name: some_name

Figure 4 – screenshot of level-redactor

This level-redactor's source code is based on the text-file generation tutorial from Tutorialzine.com [12].

2. Theory

The concept of 4D space implemented in this work is purely geometrical, which means that all the four dimensions are spatial.

2.1 Ana and Kata

Moving along every coordinate axis could be performed in two directions. In three-dimensional world those directions are called left and right, forward and backward, up and down.

But in case of four-dimensional space there is a need in an additional pair of words to describe the two directions of movement along the fourth coordinate axis. The names of those two directions are ana and kata [7].

2.2 Representation of Four-Dimensional Space

Since a human being can visually imagine only three spatial dimensions, any graphical representation of a 4D world should be limited in one way or another. Since human intuition is confined to natural three dimensions, the graphical representation of a four-dimensional space can only operate with three dimensions, which means that one of the dimensions should be reduced during the visualisation process.

This project is an engine for a computer game, the main requirement for the project is player's entertainment. Therefore, a method for reducing a dimension should be easily understandable. A method of three-dimensional section is chosen.

2.3 Three-Dimensional Section

In case of N-dimensional space, a section of N-dimensional object is an (N - 1)-dimensional object. In case of a three-dimensional object, its section is planar.

For instance, a sphere's section is circle. Depending on the place and angle in which the section was performed, the resulting circle-shaped section will be different in each case. Therefore, generally, numerous sections of a single object will have different shapes. It leads to a conclusion, that in order to perceive the N-dimensional world using only (N - 1) dimensions, the place of the section should change along the reduced direction’s basis-vector.

This principle will be applied to the 4D objects during visual representation of the 4D world: since a section of a 4D object is a volume, three-dimensional section of a 4D world will be rendered. Observing 4D world can be achieved by changing the fourth coordinate of a section volume: the section will change depending on the fourth coordinate of the section volume.

2.4 Limitations

In order for the gaming process to be entertaining and easily understandable by the player, the four-dimensional world model has following limitations:

- The section volume can only move along the fourth dimension basis-vector, no rotation of a section volume allowed.
- Moving in fourth dimension is performed with fixed steps (fourth coordinate is an integer).

Additionally, the gaming process has a limitation that any moving object can only exist at a single fourth coordinate at a particular moment because of player’s ability to observe only three dimensions at each moment. Otherwise, events such as collision with currently non-visible objects will occur.

2.5 Player’s Point of View

When the principles and limitations described above are applied, the picture af a 4D world from the player’s point of view at a particular moment of time should look exactly like a three-dimensional world. But because of the player’s ability to change his own fourth coordinate, the world observed will change its look in time according to the player’s current fourth coordinate.

3. General Principles of Implementation

3.1 Concept of Four-Dimensional Objects

The position of any object in N-dimensional space can be described by a set of N coordinates that stores the location of the object's center point on each coordinate. The location of an object in 4D space can be represented by four numbers.

The first three coordinates (X, Y, Z) of each object are floating-point numbers because some objects have ability to move and such precision is required in order to achieve the effect of smooth movement.

Because of the limitation for each object to have an integer for its fourth coordinate (W), it is stored apart from the other three coordinates. Another reason for this decision is that movement in W coordinate is performed less frequently than in other axes. Finally, no smooth movement along W axis is required.

To sum up, the position of objects in 4D space is stored in two separate structures: first structure for X, Y and Z coordinates, second structure for W coordinate.

3.2 Rendering and Interaction

The chosen 4D world representation can be described as a set of coexisting three-dimensional layers. The W-coordinate of an object represents the layer where the object is located and the first three coordinates represent the location of an object in this layer.

Since the player has single W-coordinate, only the volumetric layer with the same W-coordinate should be rendered.

The same principle is applied for any interaction between objects. For example, collision is only performed for objects that are located in the same layer.

3.3 Source Code Examples Used

Particular parts of the source code were based on the source codes of the tutorials written by Sri Harsha Chilakapati. The sources were taken from his public repository at github.com [11].

Most of the utility classes were not changed or only additional methods were created. Some major classes were based on the short code examples from the tutorials. Some classes and methods from the tutorial code were significantly modified.

In each case of code adoption the initial author is mentioned in the comments of the source code of the game.

4. Game Loop

The game loop is a cycle that begins after all the resources are loaded in `init()` method. There are three main responsibilities of the game loop.

Firstly, `update(...)` method is called. It updates the current game state: changes positions of objects, detects collisions, handles player's inputs, etc.

Secondly, the resulting picture of a current game state is being prepared in `render()` method to be shown in the next step.

Finally, `update()` method of the LWJGL `Display` class is called, which results in the image of the current game state to be shown.

The game process lasts until the execution of the main loop is ended.

```
init();
     while (!Display.isCloseRequested()) {
          ...
          update(...);//updates game state
          ...
          render();
          Display.update();
          ...
     }
end();
```

When the main game loop ends, all the loaded resources are destroyed in the `end()` method.

5. Hierarchy

5.1 Entities

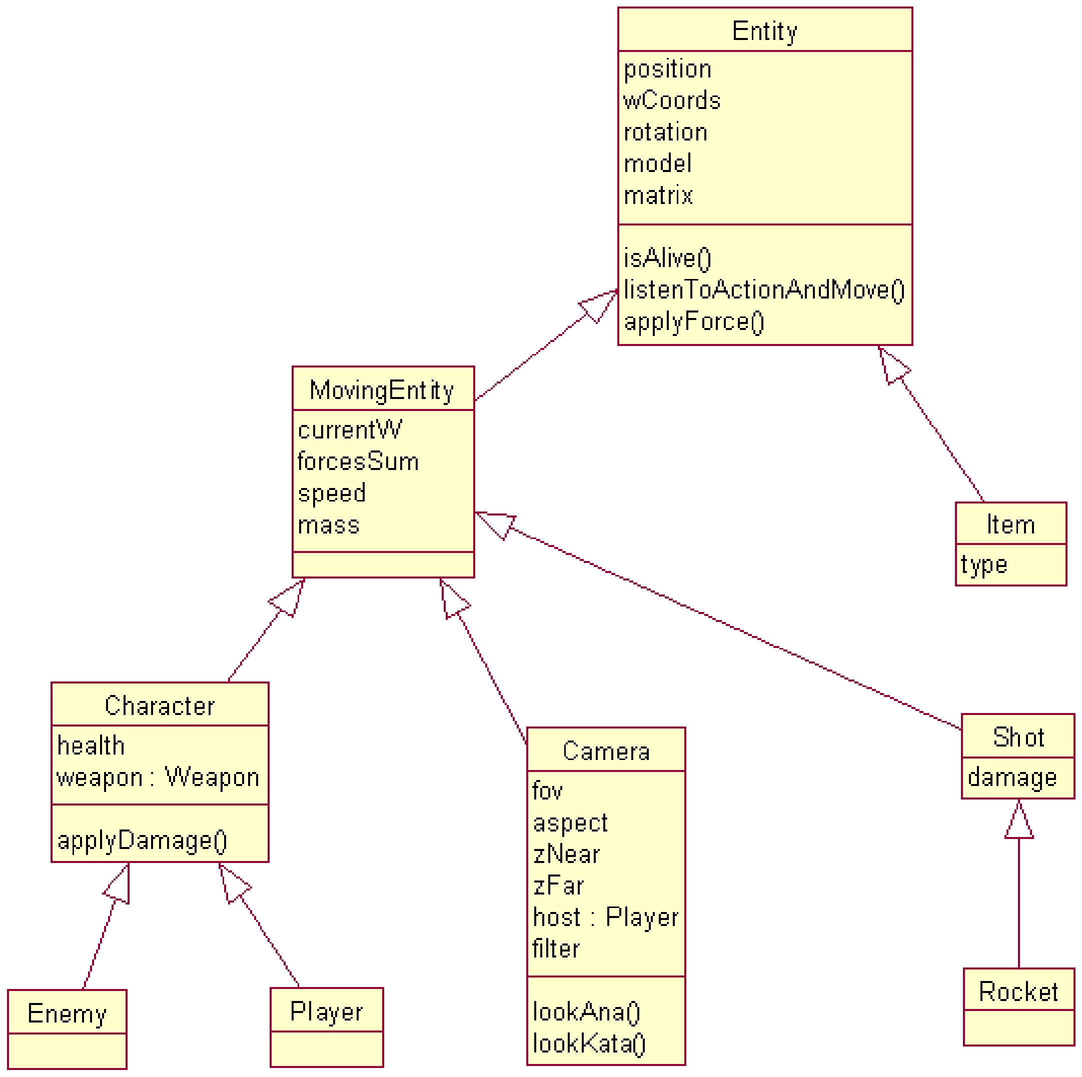


Figure 5 - Hierarchy of entities, most important fields and methods mentioned.

Every object in game that has a specific position in 4D space is an instance of `Entity`. It is a top-level class that defines majority of methods which can be optionally overridden by the classes located lower in this hierarchy (Figure 5).

For example, it is not expected for `Entity` instance to move: applying any force to `Entity` will have no effect, because `applyForce()` method's body is empty. But this method is overridden in `MovingEntity` class, so its instance will move.

This hierarchy results in every `Entity` to have specific behavior in accordance with its subtype. In this project it is useful because it provides high level of abstraction.
For example, a collection of `Entity` instances could be iterated and `isAlive()` method called on each `Entity`. If `false` is returned, then this `Entity` should be removed from the collection. Since subtypes override `isAlive()` method differently, the `Enemy` instance will be removed because its health equals zero and the `Rocket` instance will be removed because it already applied damage on some `Character` instance.
Another advantage of this hierarchy is that it is simply extendable. For example, to add a new type of `Shot` all that needs to be done is a new `Shot` subclass created.

5.2 Weapons

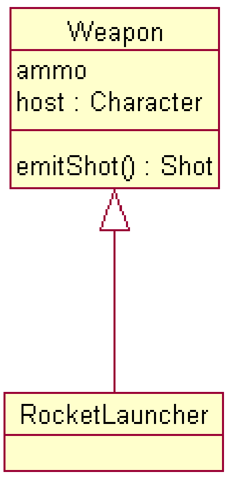


Figure 6 - hierarchy of weapons

`Weapon` (Figure 6) is not an `Entity` subtype, because its position is irrelevant and only the holder's position and rotation is considered during shooting.

6. Models

The models in project are represented in a usual for computer graphics way. Each model has a set of points called vertices. A triangle that is defined by three vertices is called a face. The surface of any model is a set of faces. But to define, for instance, an order in which the vertices should be connected to form a set of faces, more information than a set of vertex coordinates is needed. Therefore, Wavefront OBJ format is used to define models [13].

6.1 Wavefront OBJ Format

The models are represented in Wavefront OBJ file format. It is an open geometry definition file format first developed by Wavefront Technologies [13]. OBJ format is used to store the information about the following properties of a model.

- Vertices in format

```
v  x-value y-value z-value.
```

For example, coordinates of three random vertices will look like shown below.

```
v 0.437499  0.164062 -0.765626
v -0.500001 0.093750 -0.687499
v 0.273436  0.164062 -0.796875
```

- Vertex normals in format

```
vn  normal-x normal-y normal-z
```

For example, normals of the three vertices above will look like shown below.

```
vn -0.796396 0.289598 0.530930
vn 0.832844 -0.378567 0.403805
vn -0.789472 -0.315790 0.526318
```

- Faces in format

```
f v1-ind//n1-ind v2-ind//n2-ind v3-ind//n3-ind
```

For example, a face using vertices and normals defined above will look like shown below.

```
        f 1//1 2//2 3//3
```

Additionally, the information about the material that is used by a model is stored. The material library `.mtl` file that the object uses is referenced by `mtllib` keyword. A particular material from the material library that is used by a set of faces is set by `usemtl` keyword [13].

The material library file stores one or more material definitions. These definitions include material color and other attribues. In this project only the color attribute is used.

The unused in this project information that can be stored in OBJ file is not mentioned.

The sample models for the game were made using 3DS MAX and Blender.

6.2 Model Factory

In situation when multiple instances of the same model have to be created, the rational way to do this is to load the information about model surface once for each model. For this purpose a `ModelFactory` class is created (Figure 7).

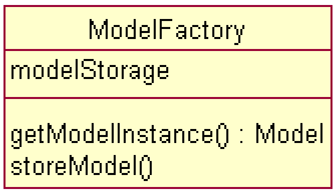


Figure 7 - main fields and methods of `ModelFactory` class

The `ModelFactory` holds a `modelStorage` map where keys are paths to OBJ files and values are `Model` instances.

The only public method of `ModelFactory` is `getModelInstance(String path).`

```
public static Model getModelInstance(String path) {
     if(modelStorage.containsKey(path)) {
          return modelStorage.get(path);
     } else {
          storeModel(path);
          return getModelInstance(path);
     }
}
```

In case of first occurrence of some path to OBJ file, the new model is instantiated and put into `modelStorage` collection. If some OBJ file with the same path has been loaded earlier, the model is reused.

`ModelFactory` guarantees that `Model` instances are reused when possible.

7. Rendering Pipeline

A raster image is a grid of pixels viewable via a monitor [14]. The process of convertation of a shape described in a vector graphics format into a raster image is called rasterization [15].

Rendering pipeline (Figures 8, 9) refers to the sequence of steps used to create a two-dimensional raster representation of a scene [16].

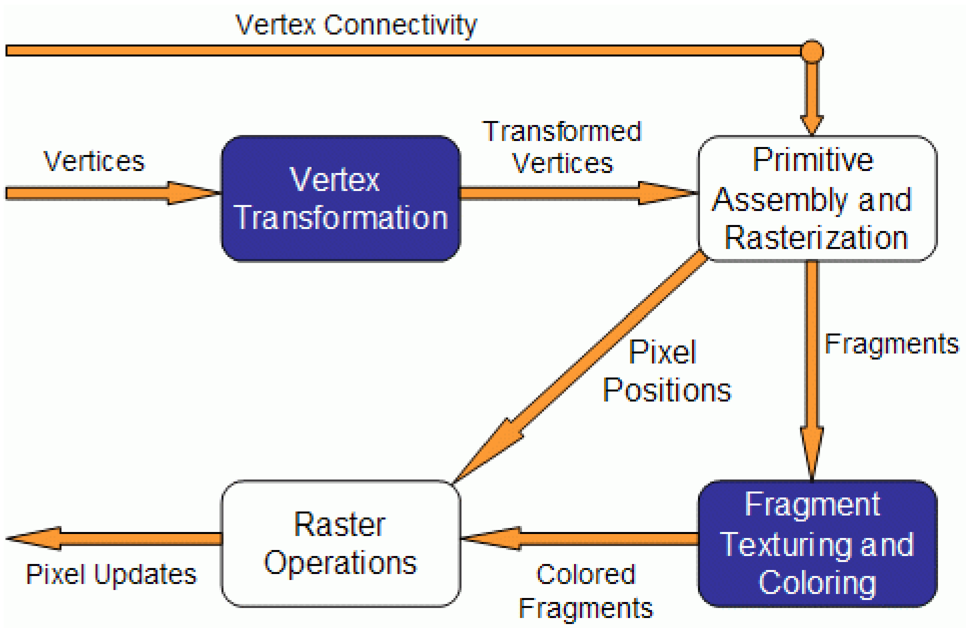


Figure 8 - Pipeline stages [17]

After the completion of all the pipeline stages the initial information about the vertices is transformed into a raster image.
On vertex transformation stage the positions of vertices are transformed and light is computated for each vertex. Vertex transformation stage is performed in vertex shader.
Then the vertices are assembled using vertex connectivity information (for example, the order in which the vertices should be assembled) and rasterization is applied. The result of this stage is a set of pixels.
On fragment texturing and coloring stage fragment coloring and texturing happens. This stage is performed in fragment shader.

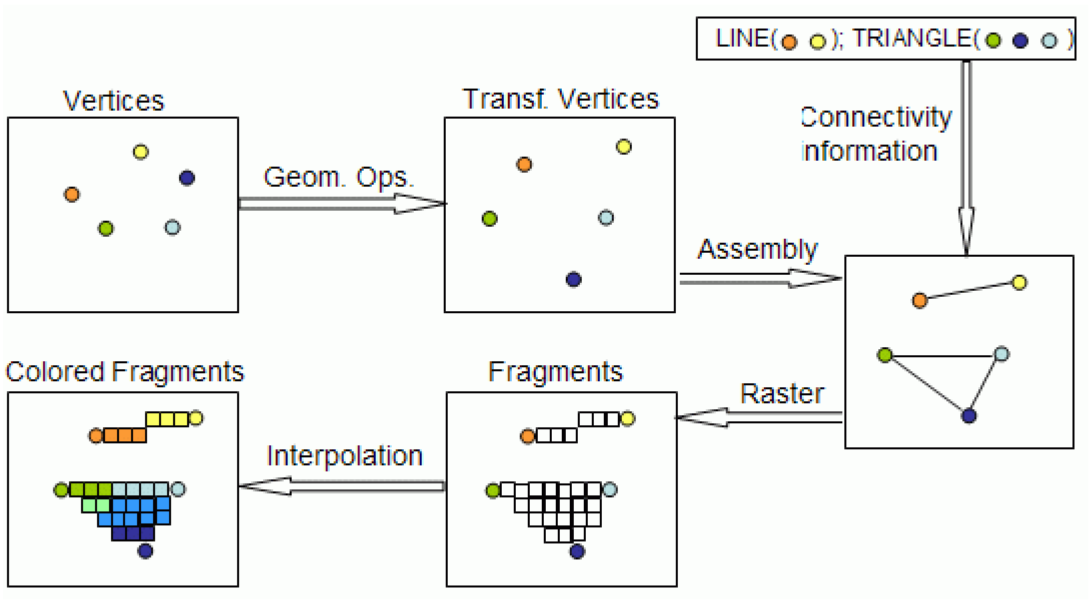


Figure 9 - Pipeline illustrated [17]

7.1 Transformations

As it has already been mentioned, every `Model` has a set of vertices. Coordinates of each vertex represent relative location of the vertex to some point of this model, usually, its center. Each `Model` instance belongs to an `Entity` instance that has position and rotation in relation to the center of the world. In order for model to be properly positioned in the world, all its vertices should be transformed using the information about the desired position and location of this model stored in `Entity` instance. This is achieved by using so-called model matrix, that is calculated in `recalculateMatrix()` method that uses the position and rotation of an `Entity`. Model matrix multiplication by each vertex coordinate of the model results in all the vertices of a model being translated to the correct location in the world. [18]

Another transformation that has to be done should consider the viewer's position and rotation in the world. Similar principle is applied using `Camera` position and rotation to calculate so-called view matrix. A product of the view matrix and the model matrix multiplication stores information about where the observed model is in relation to the `Camera`. The view matrix is calculated in `getViewMatrix()` method.

Finally, there is a projection matrix that holds information about the properties of a perspective projection to be applied. The perspective projection can be imagined as a frustum (Figure 10).

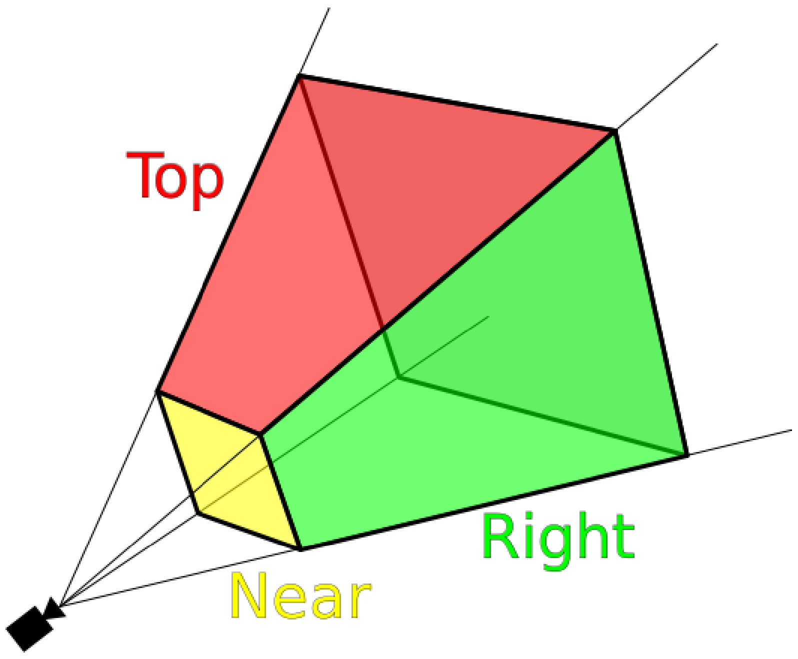


Figure 10 – the clipping planes of viewing frustum [19]

Using perspective projection will result in far objects appear smaller and near objects bigger as in real world.
OpenGL operates with coordinates from -1 to 1, so a point outside the viewing frustum will be clipped, because after applying all the transformations this point's coordinates will lie outside of [-1 … 1] range [18].
The product of projection, view and model matrices multiplication is a final transformation matrix. This tranformation matrix is multiplied by a column matrix consisting of vertex coordinates, which produces the resulting screen coordinates of a vertex that OpenGL uses during next stages of a rendering pipeline.

The projection, view and model matrices are passed to the vertex shader, where the transformations are applied for each vertex.

7.2 Vertex Shader

A vertex shader is a program that is usually executed for each vertex on a graphic processing unit of a computer's graphics card. In this project GLSL language is used to program shaders. To perform all the transformations, the input parameters such as transformation matrices are passed to the vertex shader:

```
//vertex shader
//declaration of inputs

uniform mat4 projection;
uniform mat4 view;
uniform mat4 model;
```

Then the tranformation is applied:

```
gl_Position = projection * view * model * gl_Vertex;
```

where `gl_Position` is a position of vertex.

A vertex shader can also pass other data to the fragment shader. See Appendix 1 for full vertex shader source code.

After vertex shader, a fragment shader is executed.

7.3 Fragment Shader

A fragment shader operates with fragments. It outputs a color value of a single pixel. Different visual effects can be implemented in fragment shader. For example, the effect of color filter when player looks ana or kata is implemented in fragment shader. For example, red filtering is done by mixing the image fragments with red color:

```
//color in RGBA format
vec4 red = vec4(1.0, 0.0, 0.0, 1.0);

//last parameter - extent of mixing
result = mix(result, red, 0.3);
```

As a result, the image is filtered (see Figure 2).

The effect of a smooth filter transition when the player moves ana or kata is achieved by changing the extent of mixing. It is calculated in Java code for every frame and passed to the fragment shader as the `float` variable `filterTransition`.

Another effect that is implemented in fragment shader is fog.

```
//define fog color (RGBA)
vec4 fog = vec4(0.46, 0.82, 0.93, 1.0);
float fogDensity = 0.05;
const float LOG2 = 1.442695;

float fogFactor = 1 - exp2(
          -fogDensity * fogDensity
          * vPosition.z *
          vPosition.z * LOG2);

//limit value to [0...1]
fogFactor = clamp(fogFactor, 0.0, 1.0);

result = mix(result, fog, fogFactor);
```

An exponential equation for fog density calculation has been applied [20, 21].

Lighting is calculated in fragment shader as well. For full fragment shader source code see Appendix 2.

8. Physics

To perform physics update, the `updatePhysics(...)` method is called from `update(...)` method of the main game loop mentioned earlier.

Inside `updatePhysics(...)` method all the `Entity` instances are iterated and their positions changed according to the vector sum of all the forces applied to a current `Entity` instance.

Then collision detection is performed for each pair of objects possible. In case of positive result the collision gets handled. See Appendix 3 for full source code of `updatePhysics(...)` method.

8.1 Axis-Aligned Bounding Box

Usually, a computer game physics model operates with so-called bounding volumes of objects when handling physical interaction. A bounding volume is a simplified version of the model. This simplification is performed to increase the performance of the application.

An axis-aligned bounding box (Figure 11) is a simple type of a bounding volume. It bounds an object's model within the box of a minimum size possible. Axis-aligned means that edges of the box are parallel to the coordinate axes [22].

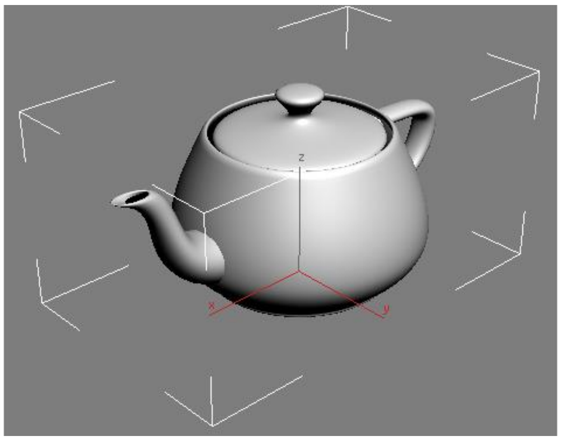


Figure 11 - a model of teapot, its bounding box and coordinate axes

In this project `calculateBB()` method of `Model` class is called after the loading of the model from OBJ file. It calculates bounding box of a model automatically by iterating all the vertices and storing maximum and minimum values of each coordinate.

8.2 Collision Detection

To determine whether it is needed to apply physical laws during interaction between objects, a collision detection algorithm is used. A `detect()` method in `Collision` class serves this purpose.

```
public boolean detect(Entity entity1, Entity entity2) {
     pos1 = entity1.getCenterPoint();
     pos2 = entity2.getCenterPoint();
     float diffX = Math.abs(pos1.x - pos2.x);
     float diffZ = Math.abs(pos1.z - pos2.z );
     float diffY = Math.abs(pos1.y - pos2.y);

     bb1 = entity1.getModel().getBoundingBox();
     bb2 = entity2.getModel().getBoundingBox();

     if(diffX < (bb1.getHalfX() + bb2.getHalfX()) ) {
          if(diffZ < (bb1.getHalfZ() + bb2.getHalfZ()) ) {
               if(diffY < (bb1.getHalfY() + bb2.getHalfY())) {
                    return true;
               }
          }
     }
     return false;
}
```

As seen from the code, the algorithm is based on simple coordinate checks. The bounding boxes should have intersection on each coordinate for collision to be detected.

8.3 Collision Handling

In case of collision, `collide(...)` method class is called. Firstly, it reduces interpenetration of the objects through each other. Then, it applies little rebound force. Finally, the law of conservation of momentum is applied.

9. Head-Up Display

A head-up display (Figure 12) is a common way of indicating game state in first-person shooters using on screen indicators. Different types of head-up display indicators were made for this project.

- Crosshair. Indicates direction where a rocket is launched. Located in the center of the screen.
- W-energy bar. Indicates how much energy player has to perform movement in W-axis. Located at the bottom of the screen.
- Health bar. Indicates how much health player has. An initiallty green bar that gradually changes its color to red depending on player’s health. Displayed above the W-energy bar.
- Ammo. Amount of rockets shown in the top left corner.
- Ana/kata arrows indicate the directions of the fourth dimension in which the world exists. Location: bottom left corner.

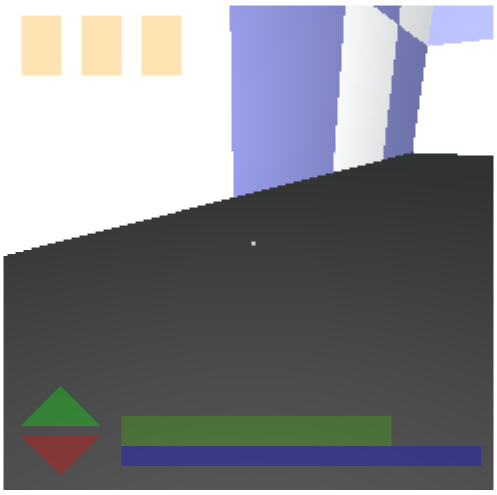


Figure 12 - all the head-up display elements shown

OpenGL has functionality of drawing simple two-dimensional shapes of chosen color and opacity on screen. For example, ana arrow drawing method is shown below.

```
private void drawAnaArrow() {
     //return if element drawing is currently not needed
     ...
     //color in RGBA (red, green, blue, alpha) format
     GL11.glColor4f(0f, 1f, 0f, DEF_ALPHA);

     GL11.glBegin(GL11.GL_TRIANGLES);
          glVertex2f(w + 10,  h + 10);//x, y coords
          glVertex2f(w + 50,  h + 10);
          glVertex2f(w + 30,  h + 30);
     GL11.glEnd();
}
```

`GL11.glBegin(...)` selects shape and tells OpenGL to start drawing it. A triangle is chosen in the example. Then the coordinates of the points of this two-dimensional shape are defined in `glVertex2f(int x, int y)` function. Finally, `GL11.glEnd()` function is called to exit from shape drawing mode.

Every other indicator is drawn in the same way, only color, opacity and shape differs.

10. Summary

The main goal of this work was to create a four-dimensional first-person shooter game prototype.

The main problems I dealt with are finding the way of representing a four-dimensional world on the screen, implementing game logic, rendering pipeline, basic physics algorithms.

The result is a game with all the basic aspects described above implemented. The following conclusions were made during the development process.

Even a prototype of a game with basic functionality and simple logic should have a logical structure. This structure should have multiple components having different responsibilities.

It is useful to apply principles of object-oriented programming for game development, because it provides developer with high level of abstraction and makes the code structure easily extendable.

In conclusion I would like to mention several techniques of developing this project further. Support of textures and bump mapping could be added to improve visual appearance of the game. A variety of methods to improve performance of the application could be implemented: for instance, optimization algorithms for physics module such as binary space partitioning [23] or more advanced octree [24] could be applied. Finally, a more sophisticated game logic could be implemented for gaming process to become more entertaining.

11. Kokkuvõte

Selle töö põhiline eesmärk oli neljamõõtmelise tulistamismängu prototüübi loomine.

Põhilised probleemid millega tegelesin oli neljamõõtmelise maailma representeerimine ekraanil, mänguloogika loomine, visualiseerimise etappide teostamine, lihtsate füüsikaalgoritmide loomine.

Tulemuseks on arvutimäng kus on kõik mainitud aspektid ellu rakendatud. Järgmised järeldused on tehtud arenduse etapi jooksul.

Isegi väikese funktsionaalsuse ja lihtsa loogikaga mängu prototüübil peab olema loogiline struktuur. Selles struktuuris peab olema erinevate vastutustega komponentide hulk.

Mangu arendamisel on kasulik objektorienteeritud programmeerimise priintsiibid kohaldada, sellepärast et need annavad arendajale kõrge abstraktsiooni tasemet ning teevad koodistruktuuri kergesti laiendatavaks.

Kokkuvõtteks mainiks ära mõned projekti edasiarenendamise viisid. Tekstuuride toetus ja “bump mapping” tehnoloogia võiks olla rakendatud mängu visuaalse külje paranemiseks. Rakenduse jõudluse tõstamise meetodid võiksid olla kasutatud. Näiteks, füüsika mooduli optimeerimisalgoritme nagu “binary space partitioning” [23] või rohkem edasijõudnud “octree” [24] võiks rakendada. Lõpuks, keerulisem mängu loogika voiks olla loodud, et mängu protsess saaks lõbustavamaks.

12. References

[1] OpenGL explanation [WWW]

[http://en.wikipedia.org/wiki/OpenGL] (accessed 04.06.2014)

[2] LWJGL explanation [WWW]

[http://en.wikipedia.org/wiki/Lightweight_Java_Game_Library] (accessed 04.06.2014)

[3] GLSL explanation [WWW]

[http://en.wikipedia.org/wiki/OpenGL_Shading_Language] (accessed 04.06.2014)

[4] Shader explanation [WWW]

[http://en.wikipedia.org/wiki/Shader] (accessed 04.06.2014)

[5] Vertex shader, fragment shader explanation [WWW]

[http://en.wikipedia.org/wiki/Shader#Types] (accessed 04.06.2014)

[6] Rendering pipeline explanation [WWW]

[http://en.wikipedia.org/wiki/Graphics_pipeline] (accessed 04.06.2014)

[7] Ana / kata meaning [WWW]

[http://en.wikipedia.org/wiki/Four-dimensional_space#Orthogonality_and_vocabulary] (accessed 04.06.2014)

[8] Video game industry worldwide revenue [WWW]

[http://vgsales.wikia.com/wiki/Video_game_industry] (accessed 04.06.2014)

[9] Game “Minecraft” [WWW]

[http://en.wikipedia.org/wiki/Minecraft] (accessed 04.06.2014)

[10] Game “Papers, Please” [WWW]

[http://en.wikipedia.org/wiki/Papers,_Please] (accessed 04.06.2014)

[11] Sri Harsha Chilakapati public source code repository [WWW]

[https://github.com/sriharshachilakapati] (accessed 07.04.2014)

[12] Text file generation tutorial [WWW]

[http://tutorialzine.com/2011/05/generating-files-javascript-php/] (accessed 04.06.2014)

[13] Wavefront OBJ format [WWW]

[http://en.wikipedia.org/wiki/Wavefront_.obj_file] (accessed 04.06.2014)

[14] Raster graphics explanation [WWW]

[http://en.wikipedia.org/wiki/Raster_graphics] (accessed 04.06.2014)

[15] Rasterization explanation [WWW]

[http://en.wikipedia.org/wiki/Rasterisation] (accessed 04.06.2014)

[16] Rendering pipeline explanation [WWW]

[http://en.wikipedia.org/wiki/Graphics_pipeline] (accessed 04.06.2014)

[17] Rendering pipeline images and overview [WWW]

[http://www.lighthouse3d.com/tutorials/glsl-tutorial/pipeline-overview/]
(accessed 04.06.2014)

[18] Projection, view, model matrices explanation [WWW]

[http://solarianprogrammer.com/2013/05/22/opengl-101-matrices-projection-view-model/]
(accessed 04.06.2014)

[19] Viewing frustum picture [WWW]

[http://en.wikipedia.org/wiki/Viewing_frustum] (accessed 04.06.2014)

[20] Fog density exponential equation [WWW]

[http://www.ozone3d.net/tutorials/glsl_fog/p02.php] (accessed 04.06.2014)

[21] Fog implementation example in GLSL [WWW]

[http://www.ozone3d.net/tutorials/glsl_fog/p04.php] (accessed 04.06.2014)

[22] Axis-aligned bounding box explanation [WWW]

[http://en.wikipedia.org/wiki/Axis-aligned_bounding_box#Axis-aligned_minimum_bounding_box] (accessed 04.06.2014)

[23] Binary space partitioning explanation [WWW]

[http://en.wikipedia.org/wiki/Binary_space_partitioning] (accessed 04.06.2014)

[24] Octree explanation [WWW]

[http://en.wikipedia.org/wiki/Octree] (accessed 04.06.2014)

Appendix 1

```
//author: Jevgeni Krutov
//author: Sri Harsha Chilakapati







uniform mat4 projection;
uniform mat4 view;
uniform mat4 model;

varying vec4 vColor;
varying vec3 vPosition;
varying vec3 vNormal;
uniform mat4 mNormal;




void main()
{
  // -> to fragment shader
  vColor = gl_Color;

  //-> to fragment shader
  vPosition = (view * model * gl_Vertex).xyz;

  //-> to fragment shader
  vNormal = normalize(mNormal * vec4(gl_Normal, 1.0)).xyz;

  gl_Position = projection * view * model * gl_Vertex;
}
```

Appendix 2

```
//author: Jevgeni Krutov
//author: Sri Harsha Chilakapati
//fog implementation based on example from
//http://www.ozone3d.net/tutorials/glsl_fog/p04.php




varying vec4 vColor;
varying vec3 vPosition;
varying vec3 vNormal;
uniform vec3 lightPos;
uniform mat4 view;



const vec3 lightColor = vec3(1.0, 1.0, 1.0);
const float lightIntensity = 2.0;
const float ambientCoefficient = 0.05;
const float shininess = 128.0;



uniform int filter;
//0 no filter, 1 green, -1 red, 2 gray


uniform float filterTransition;
//0 no transition, 1 - green, -1 red
```

```
void main()
{
    vec3 surfaceToLight = normalize(
          vec3(view * vec4(lightPos, 1.0)) - vPosition);


    vec3 ambient =
          ambientCoefficient * vColor.rgb * lightIntensity;


    float diffuseCoefficient =
          max(0.0, dot(vNormal, surfaceToLight));


    vec3 diffuse =
          diffuseCoefficient * vColor.rgb * lightIntensity;


    float specularCoefficient = 0.0;


    if(diffuseCoefficient > 0.0) {
        specularCoefficient =
          pow(max(0.0, dot(surfaceToLight,
               reflect(-surfaceToLight, vNormal))),
               shininess);
    }


    vec3 specular =
          specularCoefficient * vec3(1.0, 1.0, 1.0) *
          lightIntensity;



    vec4 result =
          vColor + vec4(ambient, 1.0) + vec4(diffuse, 1.0) *
          vec4(lightColor, 1.0) + vec4(specular, 1.0);
```

```
//---FOG---

vec4 fog = vec4(0.46, 0.82, 0.93, 1.0);
float fogDensity = 0.05;

const float LOG2 = 1.442695;

float fogFactor = 1 - exp2(
      -fogDensity * fogDensity *   vPosition.z *
      vPosition.z * LOG2);

fogFactor = clamp(fogFactor, 0.0, 1.0);

result = mix(result, fog, fogFactor);




//---FILTERING---

vec4 red = vec4(1.0, 0.0, 0.0, 1.0);
vec4 green = vec4(0.0, 1.0, 0.0, 1.0);
vec4 white = vec4(1.0, 1.0, 1.0, 1.0);

if(filter == 0) {
      //no filtering, but transition may be present

      if(filterTransition > 0f) {
           //green transition
           result = mix(
                result, green,
           0.3*abs(filterTransition));
      }

      if(filterTransition < 0f) {
                //red transition
                result = mix(result, red,
      0.3*abs(filterTransition));
      }

      if(filterTransition > 0.9f || filterTransition < -
      0.9f) {
                result = mix(
                     result, white,
                0.7*pow(filterTransition, 2f));
      }

      gl_FragColor = result;
```

```
    } else if(filter == 1) {
         //green filtering
         gl_FragColor = mix(result, green, 0.3);

    } else if(filter == -1) {
         //red filtering
         gl_FragColor = mix(result, red, 0.3);

    } else if(filter == 2) {

         //gray filtering
         float mono = (result.x + result.y + result.z) / 3;
         gl_FragColor = vec4(mono, mono, mono, 1.0);
    }

} //-main
```

Appendix 3

```
private void updatePhysics(long elapsedTime) {
     for(int i=0; i < level.entities.size(); i++) {
          Entity eI = level.entities.get(i);

          if(!eI.isShot())
               eI.applyForce(
                    level.getGravity().getGravityVector());

          eI.changeSpeedAndMove(elapsedTime);

          for(int j = i + 1; j < level.entities.size(); j++){

               Entity eJ = level.entities.get(j);

               if(!eI.isMovingEntity()
                    && !eJ.isMovingEntity()) {
                    //at least one in a pair should be
                    //MovingEntity to perform collision-test
                    continue;
               }

               if(!level.getCollisionDetector()
               .containSameW(eI, eJ)) {
                    //both entities should have at least
                    //one common w to
                    //perform collision-test
                    continue;
               }

               if(level.getCollisionDetector()
               .detect(eI, eJ)) {
               //collision occurred
               level.getCollisionDetector()
                    .collide(eI, eJ);
               }
          }
     }
}
```