Asteroids

This game is based on the classic 2D asteroids game (<https://en.wikipedia.org/wiki/Asteroids_(video_game)>). However, this version is in 3D. In this game, you control a ship and shoot different asteroids. When you get hit, the game ends. The more asteroids you shoot, the more points you get. An extra challenge is also added in the form of other ships which will try and shoot you down. Each ship has an AI that controls its movements. It uses the OpenGL graphics library to create a new window which vertices are drawn onto and connected to form triangles and allows me to add listeners for keyboard and mouse inputs easily. I explored the use of using the GPU as a general processor and not just for graphics too. Originally, I was going to use blender to create complex shapes like ships and asteroids and create a .obj reader (using the Wavefront format) but instead I have used a Perlin noise algorithm to procedurally generate each asteroid mesh. The ship is made up of several simpler objects (2 cuboids and a prism) which I have grouped together.

# Features

|  |  |  |
| --- | --- | --- |
| Feature | Achieved | Technical skills |
| 3D graphics | Yes | * Complex mathematical model * Advanced vector and matrix operations * Use of GPU parallelisation |
| Collision detection (Separating axis theorem) | Yes (apart from last point which is only partially complete). Currently only the basic java implementation is finished | * Complex mathematical model * Advanced vector operations * Use of GPU parallelisation capabilities to boost performance * Complex user defined algorithm |
| Projectile motion (for bullets) | Yes | * Complex mathematical model * Advanced vector operations |
| Quicker world coordinates processor (i.e. transforming each model coordinate to world coordinate on a GPU instead of CPU) | Yes | * Advanced matrix operations * Complex mathematical model * Complex user defined algorithm * Use of GPU parallelisation |
| Procedurally generated asteroids (using Perlin noise to define the surface of an asteroid) | Yes | * Complex user defined algorithm * Complex mathematical model |
| Dynamically spawn and despawn different objects (each type of object (e.g. ship, asteroid, bullet) is its own class which will extend abstract classes like ObjectAssembly, RigidObject. Those objects also have parent objects like GameObject or NonRenderableObject. All types of objects extend NonRenderableObject) | Yes | * Complex user-defined OOP programming model with inheritance, polymorphism, interfaces, classes |
| Save game | No | * Reading and writing to the file system |
| Use of Java Generics (for a utility class that I used to send data to the renderer | Yes | * Complex OOP model (generics) |
| Use of interfaces (ILogic) | Yes | * Complex OOP model (interfaces) |
| Ship AI | No | * Use of graphs/trees * Recursive algorithms |
| Broke down complex functions (like GPU implementation of SAT) into smaller functions and used a pipeline | Yes | * Complex user defined model * Decomposition/Composition |
| Read file in jar (e.g. texture or OpenCL/OpenGL shader code) | Yes | * Reading data from the file system |

Other technical skills used in these features include:

* List operations

# Code

## Net.alevel.asteroids.engine

### GameEngine.java

**package** net.alevel.asteroids.engine;

**import** java.util.List;

**import** net.alevel.asteroids.engine.graphics.Camera;

**import** net.alevel.asteroids.engine.graphics.Renderer;

**import** net.alevel.asteroids.engine.input.Input;

**import** net.alevel.asteroids.engine.objects.GameObject;

**import** net.alevel.asteroids.engine.utils.Pair;

/\*\*This contains the actual game loop. It calls different functions in {@link ILogic} which causes the simulation to do something

\* Most of the classes in engine.graphics and the game loop (which i modified a little) are based off this tutorial ->

\* <a href="https://ahbejarano.gitbook.io/lwjglgamedev">https://ahbejarano.gitbook.io/lwjglgamedev</a>

\*/

**public** **class** GameEngine **implements** Runnable {

/\*\*The target number of 'Frames per second' to be rendered.

\*/

**public** **static** **final** **int** ***TARGET\_FPS*** = 60;

/\*\*The target number of 'Updates per second'. It is the speed of the in game clock.

\*/

**public** **static** **final** **int** ***TARGET\_UPS*** = 100; //updates per second

**private** **final** Window window;

**private** **final** Renderer renderer;

**private** **final** ILogic gameLogic;

**private** **final** Input humanInput;

**public** GameEngine(ILogic gameLogic) {

**this**.window = **new** Window();

**this**.renderer = **new** Renderer();

**this**.gameLogic = gameLogic;

**this**.humanInput = **new** Input();

}

/\*\*Starts the simulation

\*/

@Override

**public** **void** run() {

**try** {

**this**.init();

**this**.gameLoop();

} **catch** (Exception e) {

System.***out***.println("A fatal error has occured and the program has stopped!");

e.printStackTrace(); //print any error to console

}

}

/\*\*Initialises the core components (like the renderer and the input sampler) and runs {@link ILogic#init(Window)} to setup the initial state of the simulation

\* **@throws** Exception

\*/

**protected** **void** init() **throws** Exception { //any errors will passed to the method that called this method

//CLManager.init();

**this**.window.init();

**this**.renderer.initShaderProgram();

**this**.humanInput.init(**this**.window);

**this**.gameLogic.init(**this**.window);

}

/\*\*This is the main game loop. Methods are protected for convenience. It may come in useful if I need to alter what happens in the loop.<br>

\* It is designed to hit {@link GameEngine#TARGET\_FPS} and {@link GameEngine#TARGET\_UPS} values. If a render takes more time than the UPS time interval, multiple updates will

\* happen before the next render to catch up. This ensures that the actual speed of the in game clock does not change when the FPS changes

\* **@throws** Exception

\*/

**protected** **void** gameLoop() **throws** Exception { //the main loop

**float** lastLoop = System.*nanoTime*() / 1000\_000\_000f; //stores time that last loop started

**float** accumulator = 0f; //stores the amount of time that the game needs to catch up with

**float** interval = 1f / ***TARGET\_UPS***; //the time interval between each update (the speed of the in game clock)

**float** loopSlot = 1f / ***TARGET\_FPS***; //The loop runs every frame per second, not every update per second

**float** totalTime = 0; //the actual in game time (time the simulation has been running for)

**while**(!**this**.window.windowShouldClose()) { //game loop will stop if the window is about to close (i.e. if the user closes the window). This will cause the whole app to terminate

**float** time = System.*nanoTime*() / 1000\_000\_000f;

accumulator += time - lastLoop; //get time (in seconds) to complete last loop and add it to the time accumulated (time behind).

lastLoop = time; //last loop is now equal to the time that this run started

**this**.input();

**for**(; accumulator >= interval; accumulator -= interval) { //keep updating until caught up with the time lost. This should mean the UPS should not change when the FPS changes

**this**.update(totalTime + accumulator, 0.0001f); //the value passed here is 1 in game time second. You can change the speed of the physics with this value

totalTime += interval; //test with this and 0.0001f

}

**this**.render(); //render

**double** endTime = time + loopSlot; //endTime is the start time + the minimum amount of time a loop is allowed to complete

**try** { //if the loop completed too quickly, the thread pauses to keep the FPS going beyond the target FPS

Thread.*sleep*((**long**) (endTime \* 1000) - (System.*nanoTime*() / 1000\_000)); //convert endTime to milliseconds (its in seconds)

} **catch** (InterruptedException e) {

e.printStackTrace();

} **catch** (IllegalArgumentException e) { //A lazy way to handle the fact that if the time elapsed is greater than the minimum time, the thread doesn't have to pause

}

}

**this**.cleanUp();

}

/\*\*Samples any keys pressed

\*/

**protected** **void** input() {

**this**.humanInput.input(**this**.window);

}

/\*\*Update objects (simulate physics for that instant of time)

\* **@throws** Exception

\*/

**protected** **void** update(**float** accumulatedTime, **float** interval) **throws** Exception {

**this**.gameLogic.update(accumulatedTime, interval, **this**.humanInput);

}

/\*\*Draw the updated objects onto the screen. Then the window will be called to swap frame buffers

\*/

**protected** **void** render() {

Pair<Camera, List<GameObject>> p = **this**.gameLogic.toRender();

**this**.renderer.render(**this**.window, p.getO1(), p.getO2());

**this**.window.update(); //the method will tell OpenGL to swap the old frame buffer with the new frame buffer (i.e update what is being displayed)

}

/\*\*To be run when the simulation is about to end.

\*/

**protected** **void** cleanUp() {

**this**.renderer.cleanUp();

**this**.gameLogic.cleanUp();

//CLManager.cleanUp();

}

}

### ILogic.java

**package** net.alevel.asteroids.engine;

**import** java.util.List;

**import** net.alevel.asteroids.engine.graphics.Camera;

**import** net.alevel.asteroids.engine.input.Input;

**import** net.alevel.asteroids.engine.objects.GameObject;

**import** net.alevel.asteroids.engine.utils.Pair;

/\*\*Implement this interface for classes that contribute to the main logic of the game. the functions {@link ILogic#update(float, float, Input)} and {@link ILogic#toRender()} run periodically

\* as it is constantly called by the game loop. {@link ILogic#init(Window)} only runs once (at the start) and {@link ILogic#cleanUp()} also only runs once (at the end)

\*/

**public** **interface** ILogic {

/\*\*Runs when the object is instantiated. Setup any initial states here

\*/

**public** **void** init(Window window) **throws** Exception;

/\*\*Defines what should happen each update (i.e what the simulation should do at that time) (e.g. change object states and/or positions)

\* **@param** interval the time between this update and the previous update (in terms of in game clock)

\* **@param** input the sample of the mouse and keyboard inputs at a certain instant

\*/

**public** **void** update(**float** accumulatedTime, **float** interval, Input input) **throws** Exception;

/\*\*Tells the renderer what to render and also passes the camera to use

\* **@param** camera camera to use

\* **@param** objectsToRender game objects that will be rendered. If you do not want an object to be rendered, exclude it from this list

\*/

**public** Pair<Camera, List<GameObject>> toRender();

/\*\*Runs when the object is about to be destroyed (either on shutdown or not needed anymore)

\*/

**public** **void** cleanUp();

}

### Window.java

**package** net.alevel.asteroids.engine;

**import** **static** org.lwjgl.glfw.GLFW.***GLFW\_CONTEXT\_VERSION\_MAJOR***;

**import** **static** org.lwjgl.glfw.GLFW.***GLFW\_CONTEXT\_VERSION\_MINOR***;

**import** **static** org.lwjgl.glfw.GLFW.***GLFW\_KEY\_ESCAPE***;

**import** **static** org.lwjgl.glfw.GLFW.***GLFW\_OPENGL\_CORE\_PROFILE***;

**import** **static** org.lwjgl.glfw.GLFW.***GLFW\_OPENGL\_FORWARD\_COMPAT***;

**import** **static** org.lwjgl.glfw.GLFW.***GLFW\_OPENGL\_PROFILE***;

**import** **static** org.lwjgl.glfw.GLFW.***GLFW\_PRESS***;

**import** **static** org.lwjgl.glfw.GLFW.***GLFW\_RELEASE***;

**import** **static** org.lwjgl.glfw.GLFW.***GLFW\_RESIZABLE***;

**import** **static** org.lwjgl.glfw.GLFW.***GLFW\_VISIBLE***;

**import** **static** org.lwjgl.glfw.GLFW.*glfwCreateWindow*;

**import** **static** org.lwjgl.glfw.GLFW.*glfwDefaultWindowHints*;

**import** **static** org.lwjgl.glfw.GLFW.*glfwGetKey*;

**import** **static** org.lwjgl.glfw.GLFW.*glfwGetPrimaryMonitor*;

**import** **static** org.lwjgl.glfw.GLFW.*glfwGetVideoMode*;

**import** **static** org.lwjgl.glfw.GLFW.*glfwInit*;

**import** **static** org.lwjgl.glfw.GLFW.*glfwMakeContextCurrent*;

**import** **static** org.lwjgl.glfw.GLFW.*glfwPollEvents*;

**import** **static** org.lwjgl.glfw.GLFW.*glfwSetFramebufferSizeCallback*;

**import** **static** org.lwjgl.glfw.GLFW.*glfwSetKeyCallback*;

**import** **static** org.lwjgl.glfw.GLFW.*glfwSetWindowPos*;

**import** **static** org.lwjgl.glfw.GLFW.*glfwSetWindowShouldClose*;

**import** **static** org.lwjgl.glfw.GLFW.*glfwShowWindow*;

**import** **static** org.lwjgl.glfw.GLFW.*glfwSwapBuffers*;

**import** **static** org.lwjgl.glfw.GLFW.*glfwSwapInterval*;

**import** **static** org.lwjgl.glfw.GLFW.*glfwWindowHint*;

**import** **static** org.lwjgl.glfw.GLFW.*glfwWindowShouldClose*;

**import** **static** org.lwjgl.opengl.GL11.***GL\_DEPTH\_TEST***;

**import** **static** org.lwjgl.opengl.GL11.***GL\_FALSE***;

**import** **static** org.lwjgl.opengl.GL11.***GL\_TRUE***;

**import** **static** org.lwjgl.opengl.GL11.*glClearColor*;

**import** **static** org.lwjgl.opengl.GL11.*glEnable*;

**import** **static** org.lwjgl.system.MemoryUtil.***NULL***;

**import** org.lwjgl.glfw.GLFWErrorCallback;

**import** org.lwjgl.glfw.GLFWVidMode;

**import** org.lwjgl.opengl.GL;

**import** org.lwjgl.opengl.GL11;

**import** org.lwjgl.opengl.GL30;

/\*\*This represents the window onto which the renderer draws to (this is what the human actually sees on the monitor)

\* Based off this tutorial -> <a href="https://ahbejarano.gitbook.io/lwjglgamedev">https://ahbejarano.gitbook.io/lwjglgamedev</a>

\*/

**public** **class** Window {

**private** **long** windowHandle; //ID of the window handler

**private** **int** width; //width and height variables will be used in the projection matrix

**private** **int** height;

**private** **boolean** resized;

**public** Window() {

**this**.width = 600; //width and height are measured in pixels

**this**.height = 600;

**this**.resized = **false**;

}

/\*\*Configures and opens the actual window

\*/

**public** **void** init() { //width and height variables are to do with the size (in pixels) of the window

GLFWErrorCallback.*createPrint*(System.***err***).set(); //sets the GL error stream to the system stream

**if**(!*glfwInit*()) //starts the GLFW library so OpenGL can now be interfaced with

**throw** **new** RuntimeException("Unable to start GLFW!");

//set window hints (window settings)

*glfwDefaultWindowHints*();

*glfwWindowHint*(***GLFW\_VISIBLE***, ***GL\_FALSE***); //window will stay hidden after creation

*glfwWindowHint*(***GLFW\_RESIZABLE***, ***GL\_TRUE***); //this window 'can' be resized

*glfwWindowHint*(***GLFW\_CONTEXT\_VERSION\_MAJOR***, 3);

*glfwWindowHint*(***GLFW\_CONTEXT\_VERSION\_MINOR***, 2);

*glfwWindowHint*(***GLFW\_OPENGL\_PROFILE***, ***GLFW\_OPENGL\_CORE\_PROFILE***);

*glfwWindowHint*(***GLFW\_OPENGL\_FORWARD\_COMPAT***, ***GL\_TRUE***);

GLFWVidMode vidMode = *glfwGetVideoMode*(*glfwGetPrimaryMonitor*()); //gets resolution of main monitor

//this statement creates the window and at the same time checks if creation was successful

**if**((**this**.windowHandle = *glfwCreateWindow*(vidMode.width(), vidMode.height() - 30, "Asteroids", ***NULL***, ***NULL***)) == ***NULL***)

**throw** **new** RuntimeException("Failed to create new window");

//this gets called every time the window is resized. This means the width and height will be updated

*glfwSetFramebufferSizeCallback*(**this**.windowHandle, (window, width, height) -> {

**this**.width = width;

**this**.height = height;

**this**.resized = **true**;

});

//setup a key callback. It will be called every time a key is presses, repeated or released

*glfwSetKeyCallback*(**this**.windowHandle, (window, key, scanCode, action, mods) -> {

**if**(key == ***GLFW\_KEY\_ESCAPE*** && action == ***GLFW\_RELEASE***)

*glfwSetWindowShouldClose*(window, **true**); //this will be detected in the rendering loop

});

*glfwSetWindowPos*(**this**.windowHandle, 0, 30);

*glfwMakeContextCurrent*(**this**.windowHandle); //tells OpenGL to use this window

*glfwSwapInterval*(1); //1 means it will use vsync (max FPS is monitor refresh rate)

*glfwShowWindow*(**this**.windowHandle); //everything is setup so can now open the window

GL.*createCapabilities*();

*glClearColor*(0f, 0f, 0f, 0f); //background colour (the colour where nothing is rendered) will be black

*glEnable*(***GL\_DEPTH\_TEST***); //tells OpenGL to draw the farthest vertices first and then work its way towards the camera. It has no concept of layers

GL30.*glPolygonMode*(GL11.***GL\_FRONT\_AND\_BACK***, GL11.***GL\_LINE***); //draws meshes as wireframes (triangles have no fill). This is for testing

}

/\*\*Set the background colour

\* **@param** r red value

\* **@param** b blue value

\* **@param** g green value

\* **@param** alpha brightness

\*/

**public** **void** setClearColour(**float** r, **float** b, **float** g, **float** alpha) {

*glClearColor*(r, g, b, alpha);

}

**public** **boolean** isKeyPressed(**int** keyCode) {

**return** *glfwGetKey*(**this**.windowHandle, keyCode) == ***GLFW\_PRESS***;

}

**public** **boolean** windowShouldClose() {

**return** *glfwWindowShouldClose*(**this**.windowHandle);

}

**public** **int** getWidth() {

**return** width;

}

**public** **int** getHeight() {

**return** height;

}

**public** **boolean** isResized() {

**return** resized;

}

**public** **void** setResized(**boolean** resized) {

**this**.resized = resized;

}

**public** **long** getWindowHandle() {

**return** **this**.windowHandle;

}

**public** **void** update() { //This method is what causes the pixels to change in the window

*glfwSwapBuffers*(windowHandle);

*glfwPollEvents*();

}

}

## net.alevel.asteroids.engine.graphics

### Camera.java

**package** net.alevel.asteroids.engine.graphics;

**import** org.joml.Vector3f;

/\*\*Encapsulates the position and rotation of the client camera.

\*/

**public** **class** Camera {

**private** **final** Vector3f position;

**private** **final** Vector3f rotation;

**public** Camera() {

**this**.position = **new** Vector3f();

**this**.rotation = **new** Vector3f();

}

/\*\*Note: this copies the values in the vector.

\* It does not reference the vector being passed (i.e. any changes made to the vector being passed will not affect the vector stored in this object)

\*/

**public** Camera(Vector3f position, Vector3f rotation) {

**this**.position = **new** Vector3f().set(position);

**this**.rotation = **new** Vector3f().set(rotation);

}

**public** Vector3f getPosition() {

**return** **this**.position;

}

**public** Vector3f getRotation() {

**return** **this**.rotation;

}

/\*\*Note: this copies the values in the vector.

\* It does not reference the vector being passed (i.e. any changes made to the vector being passed will not affect the vector stored in this object)

\*/

**public** **void** setPosition(Vector3f position) {

**this**.position.set(position);

}

**public** **void** setPosition(**float** x, **float** y, **float** z) {

**this**.position.x = x;

**this**.position.y = y;

**this**.position.z = z;

}

/\*\*Note: this copies the values in the vector.

\* It does not reference the vector being passed (i.e. any changes made to the vector being passed will not affect the vector stored in this object)

\*/

**public** **void** setRotation(Vector3f rotation) {

**this**.rotation.set(rotation);

}

**public** **void** setRotation(**float** x, **float** y, **float** z) {

**this**.rotation.x = x;

**this**.rotation.y = y;

**this**.rotation.z = z;

}

/\*\*Explained in more detail here -> {@link Camera#movePosition(float, float, float)}

\* **@param** delta. The vector to move by

\* **@see** Camera#movePosition(float, float, float)

\*/

**public** **void** movePosition(Vector3f delta) {

**this**.movePosition(delta.x, delta.y, delta.z);

}

/\*\*Adds these offsets to the current position vector.<br>

\* Note that this vector is rotated by the current camera rotation<br>

\* e.g. if the camera is facing towards z = -infinity then a move vector of (1,0,0) (i.e. forward) will move it in that direction rather than towards x = +infinity

\* **@param** offsetX

\* **@param** offsetY

\* **@param** offsetZ

\*/

**public** **void** movePosition(**float** offsetX, **float** offsetY, **float** offsetZ) {

**if**(offsetZ != 0) {

**this**.position.x += (**float**) Math.*sin*(Math.*toRadians*(rotation.y)) \* -1f \* offsetZ;

**this**.position.z += (**float**) Math.*cos*(Math.*toRadians*(rotation.y)) \* offsetZ;

}

**if**(offsetX != 0) {

**this**.position.x += (**float**) Math.*sin*(Math.*toRadians*(rotation.y - 90)) \* -1f \* offsetX;

**this**.position.z += (**float**) Math.*cos*(Math.*toRadians*(rotation.y - 90)) \* offsetX;

}

**this**.position.y += offsetY;

}

/\*\*Adds these offsets to the current rotation vector

\* **@param** offsetX

\* **@param** offsetY

\* **@param** offsetZ

\*/

**public** **void** moveRotation(**float** offsetX, **float** offsetY, **float** offsetZ) {

**this**.rotation.x += offsetX;

**this**.rotation.y += offsetY;

**this**.rotation.z += offsetZ;

}

}

### Mesh.java

**package** net.alevel.asteroids.engine.graphics;

**import** java.io.BufferedReader;

**import** java.io.IOException;

**import** java.io.InputStreamReader;

**import** java.util.ArrayList;

**import** java.util.List;

**import** org.joml.Vector2f;

**import** org.joml.Vector3f;

/\*\*No longer needed as I do no longer plan on using Blender to generate objects<br>

\* **@see** <a href="https://en.wikipedia.org/wiki/Wavefront\_.obj\_file">https://en.wikipedia.org/wiki/Wavefront\_.obj\_file</a>

\*/

**public** **class** WavefrontMeshLoader {

/\*\*Loads a .obj file

\* **@param** fileName {@link WavefrontMeshLoader#readAllLines(String)}

\* **@return** A mesh object

\* **@throws** ClassNotFoundException

\* **@throws** IOException

\*/

**public** **static** Mesh loadMesh(String fileName) **throws** ClassNotFoundException, IOException {

List<String> lines = *readAllLines*(fileName);

List<Vector3f> vertices = **new** ArrayList<>();

List<Vector2f> textures = **new** ArrayList<>();

List<Vector3f> normals = **new** ArrayList<>();

List<Face> faces = **new** ArrayList<>();

**for** (String line : lines) {

String[] tokens = line.split("\\s+");

**switch** (tokens[0]) {

**case** "v":

// Geometric vertex

Vector3f vec3f = **new** Vector3f(

Float.*parseFloat*(tokens[1]),

Float.*parseFloat*(tokens[2]),

Float.*parseFloat*(tokens[3]));

vertices.add(vec3f);

**break**;

**case** "vt":

// Texture coordinate

Vector2f vec2f = **new** Vector2f(

Float.*parseFloat*(tokens[1]),

Float.*parseFloat*(tokens[2]));

textures.add(vec2f);

**break**;

**case** "vn":

// Vertex normal

Vector3f vec3fNorm = **new** Vector3f(

Float.*parseFloat*(tokens[1]),

Float.*parseFloat*(tokens[2]),

Float.*parseFloat*(tokens[3]));

normals.add(vec3fNorm);

**break**;

**case** "f":

Face face = **new** Face(tokens[1], tokens[2], tokens[3]);

faces.add(face);

**break**;

**default**:

// Ignore other lines

**break**;

}

}

**return** *reorderLists*(vertices, textures, normals, faces);

}

**private** **static** Mesh reorderLists(List<Vector3f> posList, List<Vector2f> textCoordList,

List<Vector3f> normList, List<Face> facesList) {

List<Integer> indices = **new** ArrayList<>();

// Create position array in the order it has been declared

**float**[] posArr = **new** **float**[posList.size() \* 3];

**int** i = 0;

**for** (Vector3f pos : posList) {

posArr[i \* 3] = pos.x;

posArr[i \* 3 + 1] = pos.y;

posArr[i \* 3 + 2] = pos.z;

i++;

}

**float**[] textCoordArr = **new** **float**[posList.size() \* 2];

**float**[] normArr = **new** **float**[posList.size() \* 3];

**for** (Face face : facesList) {

IdxGroup[] faceVertexIndices = face.getFaceVertexIndices();

**for** (IdxGroup indValue : faceVertexIndices) {

*processFaceVertex*(indValue, textCoordList, normList,

indices, textCoordArr, normArr);

}

}

**int**[] indicesArr = **new** **int**[indices.size()];

indicesArr = indices.stream().mapToInt((Integer v) -> v).toArray();

Mesh mesh = **new** Mesh(posArr, textCoordArr, normArr, indicesArr);

**return** mesh;

}

**private** **static** **void** processFaceVertex(IdxGroup indices, List<Vector2f> textCoordList,

List<Vector3f> normList, List<Integer> indicesList,

**float**[] texCoordArr, **float**[] normArr) {

// Set index for vertex coordinates

**int** posIndex = indices.idxPos;

indicesList.add(posIndex);

// Reorder texture coordinates

**if** (indices.idxTextCoord >= 0) {

Vector2f textCoord = textCoordList.get(indices.idxTextCoord);

texCoordArr[posIndex \* 2] = textCoord.x;

texCoordArr[posIndex \* 2 + 1] = 1 - textCoord.y;

}

**if** (indices.idxVecNormal >= 0) {

// Reorder vectornormals

Vector3f vecNorm = normList.get(indices.idxVecNormal);

normArr[posIndex \* 3] = vecNorm.x;

normArr[posIndex \* 3 + 1] = vecNorm.y;

normArr[posIndex \* 3 + 2] = vecNorm.z;

}

}

**protected** **static** **class** Face {

/\*\*

\* List of idxGroup groups for a face triangle (3 vertices per face).

\*/

**private** IdxGroup[] idxGroups = **new** IdxGroup[3];

**public** Face(String v1, String v2, String v3) {

idxGroups = **new** IdxGroup[3];

// Parse the lines

idxGroups[0] = parseLine(v1);

idxGroups[1] = parseLine(v2);

idxGroups[2] = parseLine(v3);

}

**private** IdxGroup parseLine(String line) {

IdxGroup idxGroup = **new** IdxGroup();

String[] lineTokens = line.split("/");

**int** length = lineTokens.length;

idxGroup.idxPos = Integer.*parseInt*(lineTokens[0]) - 1;

**if** (length > 1) {

// It can be empty if the obj does not define text coords

String textCoord = lineTokens[1];

idxGroup.idxTextCoord = textCoord.length() > 0 ? Integer.*parseInt*(textCoord) - 1 : IdxGroup.***NO\_VALUE***;

**if** (length > 2) {

idxGroup.idxVecNormal = Integer.*parseInt*(lineTokens[2]) - 1;

}

}

**return** idxGroup;

}

**public** IdxGroup[] getFaceVertexIndices() {

**return** idxGroups;

}

}

**protected** **static** **class** IdxGroup {

**public** **static** **final** **int** ***NO\_VALUE*** = -1;

**public** **int** idxPos;

**public** **int** idxTextCoord;

**public** **int** idxVecNormal;

**public** IdxGroup() {

idxPos = ***NO\_VALUE***;

idxTextCoord = ***NO\_VALUE***;

idxVecNormal = ***NO\_VALUE***;

}

}

/\*\*Reads all lines in a file. This is used for files that are stored internally within the .jar and not for files on the file system.

\* **@param** fileName name of the file

\* **@return** list of strings where each string is a line

\* **@throws** ClassNotFoundException

\* **@throws** IOException

\*/

**public** **static** List<String> readAllLines(String fileName) **throws** ClassNotFoundException, IOException {

List<String> list = **new** ArrayList<String>();

**try**(BufferedReader br = **new** BufferedReader(**new** InputStreamReader(Class.*forName*(WavefrontMeshLoader.**class**.getName()).getResourceAsStream(fileName)))) {

String line;

**while**((line = br.readLine()) != **null**)

list.add(line);

}

**return** list;

}

}

### Renderer.java

**package** net.alevel.asteroids.engine.graphics;

**import** **static** org.lwjgl.opengl.GL11.***GL\_COLOR\_BUFFER\_BIT***;

**import** **static** org.lwjgl.opengl.GL11.***GL\_DEPTH\_BUFFER\_BIT***;

**import** **static** org.lwjgl.opengl.GL11.*glClear*;

**import** **static** org.lwjgl.opengl.GL11.*glViewport*;

**import** java.util.List;

**import** org.joml.Matrix4f;

**import** net.alevel.asteroids.engine.Window;

**import** net.alevel.asteroids.engine.objects.GameObject;

/\*\*This manages which objects are to be rendered and also manages the uniforms within the shader program

\*/

**public** **class** Renderer {//Might be worth making this a singleton as I don't see why you would want more than 1 instance of the renderer. It would just cause problems

**private** **static** **final** **float** ***FOV*** = (**float**) Math.*toRadians*(60f); //FOV value is the largest angle from the centre that you can see on the screen

**private** **static** **final** **float** ***Z\_NEAR*** = 0.01f; //closest distance before object is no longer rendered

**private** **static** **final** **float** ***Z\_FAR*** = 1000f; //farthest distance before object is no longer rendered.

**private** **final** Transformations transformations;

**private** ShaderProgram shaderProgram;

**public** Renderer() **throws** IllegalStateException {

**this**.transformations = **new** Transformations();

//this.initShaderProgram(window);

}

/\*\*Loads the shader code and compiles it and links it to OpenGL. The uniform variables (variables used to interface between GLSL shaders and Java)

\* **@throws** IllegalStateException

\*/

**public** **void** initShaderProgram() **throws** IllegalStateException {

**this**.shaderProgram = **new** ShaderProgram(); //create the shader program instance

**this**.shaderProgram.initShaders(); //initialise shaders (compile GLSL and create shader objects)

**this**.shaderProgram.link(); //apply to shader pipeline

//Create uniforms (GLSL variables that can be accessed through the OpenGL API)

**this**.shaderProgram.createUniform("projectionMatrix");

**this**.shaderProgram.createUniform("modelViewMatrix");

**this**.shaderProgram.createUniform("texture\_sampler");

**this**.shaderProgram.createUniform("colour");

**this**.shaderProgram.createUniform("useColour");

//this.shaderProgram.createUniform("lightCoord"); //light test

}

/\*Clears the whole screen

\*/

**public** **void** clear() {

*glClear*(***GL\_COLOR\_BUFFER\_BIT*** | ***GL\_DEPTH\_BUFFER\_BIT***); //| is bitwise OR. Clears the 2 buffers specified (sets them to their clear values)

}

/\*\*Renders all the renderable objects. It applies the world, camera and projection matrices to get the vertices the OpenGL should draw onto the window.<br>

\* It uses the GLSL shaders to do this.<br>

\* Each time this procedure runs, a new frame is generated

\* **@param** window Reference to the window to draw to

\* **@param** camera The client camera

\* **@param** gameObjects The objects to render

\*/

**public** **void** render(Window window, Camera camera, List<GameObject> gameObjects) {

**this**.clear();

**if**(window.isResized()) {//If window is resized, update how OpenGL transforms the final projection coordinates to pixel coordinates

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

window.setResized(**false**);

}

**this**.shaderProgram.bind(); //Methods that use the shader program will use this instance

//this.shaderProgram.setUniform("lightCoord", new Vector3f(0, 0, 0)); //light test

//Get updated projection matrix which depends on the new window size

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

**this**.shaderProgram.setUniform("projectionMatrix", projectionMatrix); //update projection matrix in shaders

//Get updated view matrix which depends on the camera position

Matrix4f viewMatrix = **this**.transformations.getViewMatrix(camera);

**this**.shaderProgram.setUniform("texture\_sampler", 0); //Only using 1 texture so set it as a constant that points to texture unit 0

//Render each game object

**for**(GameObject o : gameObjects) {

//Get updated model view matrix

Matrix4f modelViewMatrix = **this**.transformations.getModelViewMatrix(o, viewMatrix);

**this**.shaderProgram.setUniform("modelViewMatrix", modelViewMatrix); //update model view matrix in the shaders

**this**.shaderProgram.setUniform("colour", o.getMesh().getColour());

**this**.shaderProgram.setUniform("useColour", o.getMesh().isTextured() ? 0 : 1);

//Once matrices are updated, we can draw the object

o.getMesh().render();

}

//Once rendering is done, unbind so accidental modifications do not happen

**this**.shaderProgram.unbind();

}

/\*\*Run this on shutdown. It unloads the shader program.

\*/

**public** **void** cleanUp() {

**this**.shaderProgram.cleanUp();

}

}

### ShaderProgram.java

**package** net.alevel.asteroids.engine.graphics;

**import** **static** org.lwjgl.opengl.GL20.*glCreateProgram*;

**import** **static** org.lwjgl.opengl.GL20.*glGetUniformLocation*;

**import** **static** org.lwjgl.opengl.GL20.*glUniform1i*;

**import** **static** org.lwjgl.opengl.GL20.*glUniform3f*;

**import** **static** org.lwjgl.opengl.GL20.\*;

**import** java.util.HashMap;

**import** java.util.Map;

**import** java.util.Scanner;

**import** org.joml.Matrix4f;

**import** org.joml.Vector3f;

**import** org.lwjgl.system.MemoryStack;

/\*\*This class manages the shader files and all the uniforms. This is the interface between java and GLSL.

\* A shader is a function that processes vertices and calculated the colour and position of each vertex on the screen.

\* These functions are executed on the GPU

\*/

**public** **class** ShaderProgram {

**private** **final** **int** programId;

**private** **int** vertexShaderId;

**private** **int** fragmentShaderId;

**private** **final** Map<String, Integer> uniforms; //this map will hold all the uniforms

**public** ShaderProgram() **throws** IllegalStateException {

**if**((**this**.programId = *glCreateProgram*()) == 0)

**throw** **new** IllegalStateException("Could not create shader program");

**this**.uniforms = **new** HashMap<String, Integer>();

}

/\*\*This adds a uniform (defined in GLSL) to the hash map so java can

\*/

**public** **void** createUniform(String uniformName) {

**int** uniformLocation = *glGetUniformLocation*(**this**.programId, uniformName);

**if**(uniformLocation < 0)

**throw** **new** IllegalStateException("Could not find uniform: " + uniformName);

**this**.uniforms.put(uniformName, uniformLocation);

}

/\*\*Use this to set a uniform to a 4x4 floating point matrix

\*/

**public** **void** setUniform(String uniformName, Matrix4f value) {

**try**(MemoryStack stack = MemoryStack.*stackPush*()) { //MemoryStack is MemoryUtil but it auto closes

*glUniformMatrix4fv*(**this**.uniforms.get(uniformName), **false**, value.get(stack.mallocFloat(16)));

}

}

/\*\*Use this to set a uniform to a 3 dimensional float vector

\*/

**public** **void** setUniform(String uniformName, Vector3f value) {

*glUniform3f*(**this**.uniforms.get(uniformName), value.x, value.y, value.z);

}

/\*\*Use this to set a uniform to an integer

\*/

**public** **void** setUniform(String uniformName, **int** value) {

*glUniform1i*(**this**.uniforms.get(uniformName), value);

}

/\*\*Load and compile GLSL code

\*/

**public** **void** initShaders() {

**try** {

**this**.vertexShaderId = **this**.initShaderObject("vertex.vs", ***GL\_VERTEX\_SHADER***);

**this**.fragmentShaderId = **this**.initShaderObject("fragment.fs", ***GL\_FRAGMENT\_SHADER***);

} **catch** (ClassNotFoundException e) { //this should never be thrown as the class its looking for is this one

e.printStackTrace();

}

}

**private** **int** initShaderObject(String fileName, **int** shaderType) **throws** ClassNotFoundException {

**int** id;

**try** (Scanner s = **new** Scanner(Class.*forName*(ShaderProgram.**class**.getName()).getResourceAsStream("/" + fileName), "UTF-8")) { //this scanner reads the file as a stream as it is within the jar

**if**((id = *glCreateShader*(shaderType)) == 0)

**throw** **new** IllegalStateException("Error creating shader object of type: " + shaderType);

*glShaderSource*(id, s.useDelimiter("\\A").next()); //links the GLSL code to the shader object

*glCompileShader*(id); //compiles the GLSL code

**if**(*glGetShaderi*(id, ***GL\_COMPILE\_STATUS***) == 0) //gets the returned code from the GLSL compiler and checks if it compiled correctly

**throw** **new** IllegalStateException("Compilation error when compiling shader code: " + *glGetShaderInfoLog*(id) + *glGetProgramInfoLog*(**this**.programId, 1024));

System.***out***.println(*glGetShaderi*(id, ***GL\_COMPILE\_STATUS***));

*glAttachShader*(**this**.programId, id);

}

**return** id;

}

/\*\*Link this shader program to the shader pipeline. Essentially applies the program to the OpenGL engine

\*/

**public** **void** link() {

*glLinkProgram*(**this**.programId);

**if**(*glGetProgrami*(**this**.programId, ***GL\_LINK\_STATUS***) == 0)

**throw** **new** IllegalStateException("Error linking shader code: " + *glGetProgramInfoLog*(**this**.programId, 1024));

**if**(**this**.vertexShaderId != 0) //the shader objects are no longer needed after linking

*glDetachShader*(**this**.programId, **this**.vertexShaderId);

**if**(**this**.fragmentShaderId != 0)

*glDetachShader*(**this**.programId, **this**.fragmentShaderId);

*glValidateProgram*(**this**.programId);

**if**(*glGetProgrami*(**this**.programId, ***GL\_VALIDATE\_STATUS***) == 0)

System.***err***.println("A warning appeared when validating shader program: " + *glGetProgramInfoLog*(**this**.programId, 1024));

System.***out***.println(*glGetProgrami*(**this**.programId, ***GL\_LINK\_STATUS***));

}

/\*\*Use if need to run methods that should refer to this object instance.

\*/

**public** **void** bind() {

*glUseProgram*(**this**.programId);

}

/\*\*Once finished with this instance, unbind from it so accidental changes are not made to this

\*/

**public** **void** unbind() {

*glUseProgram*(0);

}

/\*\*Destroys the program completely. Use on shutdown

\*/

**public** **void** cleanUp() {

**this**.unbind();

**if**(**this**.programId != 0)

*glDeleteProgram*(**this**.programId);

}

}

### Texture.java

**package** net.alevel.asteroids.engine.graphics;

**import** **static** org.lwjgl.opengl.GL11.***GL\_RGBA***;

**import** **static** org.lwjgl.opengl.GL11.***GL\_TEXTURE\_2D***;

**import** **static** org.lwjgl.opengl.GL11.***GL\_UNPACK\_ALIGNMENT***;

**import** **static** org.lwjgl.opengl.GL11.***GL\_UNSIGNED\_BYTE***;

**import** **static** org.lwjgl.opengl.GL11.*glBindTexture*;

**import** **static** org.lwjgl.opengl.GL11.*glDeleteTextures*;

**import** **static** org.lwjgl.opengl.GL11.*glGenTextures*;

**import** **static** org.lwjgl.opengl.GL11.*glPixelStorei*;

**import** **static** org.lwjgl.opengl.GL11.*glTexImage2D*;

**import** **static** org.lwjgl.opengl.GL30.*glGenerateMipmap*;

**import** **static** org.lwjgl.stb.STBImage.*stbi\_failure\_reason*;

**import** **static** org.lwjgl.stb.STBImage.*stbi\_image\_free*;

**import** **static** org.lwjgl.stb.STBImage.*stbi\_load*;

**import** java.nio.ByteBuffer;

**import** java.nio.IntBuffer;

**import** org.lwjgl.system.MemoryStack;

/\*\*Stores a texture to be applied to a mesh. Including the ID of the texture buffer that stores the texture

\*/

**public** **class** Texture {

**private** **final** **int** id;

**public** Texture(String fileName) {

**this**(*loadTexture*(fileName));

}

**public** Texture(**int** id) {

**this**.id = id;

}

**public** **void** bind() {

*glBindTexture*(***GL\_TEXTURE\_2D***, **this**.id);

}

**public** **int** getId() {

**return** **this**.id;

}

**private** **static** **int** loadTexture(String fileName) {

**int** width;

**int** height;

ByteBuffer buf;

**try** (MemoryStack stack = MemoryStack.*stackPush*()) {

IntBuffer w = stack.mallocInt(1);

IntBuffer h = stack.mallocInt(1);

IntBuffer channels = stack.mallocInt(1);

//load the raw bytes from the specified file and stores it in the byte buffer

**if**((buf = *stbi\_load*(fileName, w, h, channels, 4)) == **null**)

**throw** **new** IllegalStateException("Error loading image '" + fileName + "': " + *stbi\_failure\_reason*());

width = w.get();

height = h.get();

}

//create texture object and bind to it as next methods will be referring to it

**int** textureId = *glGenTextures*();

*glBindTexture*(***GL\_TEXTURE\_2D***, textureId);

*glPixelStorei*(***GL\_UNPACK\_ALIGNMENT***, 1); //Tells the engine how to unpack the RGBA bytes. Each component is 1 byte (second parameter)

*glTexImage2D*(***GL\_TEXTURE\_2D***, 0, ***GL\_RGBA***, width, height, 0, ***GL\_RGBA***, ***GL\_UNSIGNED\_BYTE***, buf);

/\*Target - the type of texture

\*Level - the level of detail. Level 0 is the base image level. Level n is the nth mipmap reduction image

\*Internal format - the number of colour components in the texture

\*Width - the width (in pixels)

\*Height - the height (in pixels)

\*Border - the distance between the edge of the texture and the edge of the shape. 0 means it goes all the way to the vertex

\*Format - the format of the pixel data

\*Type - the data type of the raw data

\*Data - the byte buffer containing the actual texture bytes

\*/

*glGenerateMipmap*(***GL\_TEXTURE\_2D***); //A mipmap is a decreasing resolution set of images generated from a high detailed texture. These are automatically applied when the object is scaled

*stbi\_image\_free*(buf); //Texture is now in graphics card memory so can be removed from main memory

**return** textureId;

}

**public** **void** cleanUp() {

*glDeleteTextures*(**this**.id); //only need one command to destroy a texture

}

}

### Transformations.java

**package** net.alevel.asteroids.engine.graphics;

**import** org.joml.Matrix4f;

**import** org.joml.Vector3f;

**import** net.alevel.asteroids.engine.objects.GameObject;

/\*\*Utility class. It has functions for the 3 matrices (view/camera, modelView, projection) used to calculate the positions of mesh vertices that OpenGL should draw.<br>

\*/

**public** **class** Transformations {

**private** **final** Matrix4f projectionMatrix; //The matrix that converts the view coords to on screen coords. Creates depth perception

**private** **final** Matrix4f viewMatrix; //calculates the position of the object relative to the camera

**private** **final** Matrix4f modelViewMatrix; //Converts the model coods to in world coords by getting the rotation and scale properties as well as the actual loction of the object

**public** Transformations() {

**this**.modelViewMatrix = **new** Matrix4f();

**this**.projectionMatrix = **new** Matrix4f();

**this**.viewMatrix = **new** Matrix4f();

}

/\*\*Sets the projection matrix and the returns it

\* **@param** fov field of view. The biggest angle from the centre that should be displayed on the screen

\* **@param** width the window width

\* **@param** height the window height

\* **@param** zNear closest distance to the camera before the object should no longer be rendered

\* **@param** zFar farthest distance to the camera before the object should no longer be rendered

\* **@return** the projection matrix. Converts view coordinates to 2D screen coordinates

\*/

**public** **final** Matrix4f getProjectionMatrix(**float** fov, **float** width, **float** height, **float** zNear, **float** zFar) {

**return** **this**.projectionMatrix.setPerspective(fov, width / height, zNear, zFar);

}

/\*\*Uses the view matrix (camera) as well as the scale (size), rotation and position to create the matrix

\* to be used to calculate the position of each vertex in a mesh in relation to the camera

\* **@param** gameObject

\* **@param** viewMatrix

\* **@return** The model view matrix

\*/

**public** Matrix4f getModelViewMatrix(GameObject gameObject, Matrix4f viewMatrix) {

Vector3f rotation = gameObject.getRotation();

**this**.modelViewMatrix.set(viewMatrix)

.translate(gameObject.getPosition())

.rotateX((**float**) Math.*toRadians*(-rotation.x))

.rotateY((**float**) Math.*toRadians*(-rotation.y))

.rotateZ((**float**) Math.*toRadians*(-rotation.z))

.scale(gameObject.getScale());

**return** **this**.modelViewMatrix;

}

/\*\*This gets the camera's rotation and position vectors as a matrix that can applied to mesh vertices.<br>

\* The camera in a simulation is always at position (0,0,0) and rotation (0,0,0).<br>

\* E.g. every time you move forwards, to the renderer, the objects are moving backwards

\* **@see** Transformations#getModelViewMatrix(GameObject, Matrix4f)

\* **@return** the view matrix

\*/

**public** Matrix4f getViewMatrix(Camera camera) {

Vector3f position = camera.getPosition();

Vector3f rotation = camera.getRotation();

**this**.viewMatrix.identity();

**this**.viewMatrix.rotate((**float**) Math.*toRadians*(rotation.x), **new** Vector3f(1, 0, 0))

.rotate((**float**) Math.*toRadians*(rotation.y), **new** Vector3f(0, 1, 0)); //rotate first so it rotates over current pos

**this**.viewMatrix.translate(-position.x, -position.y, -position.z); //then translate

**return** **this**.viewMatrix;

}

}

### WavefrontMeshLoader.java

**package** net.alevel.asteroids.engine.graphics;

**import** java.io.BufferedReader;

**import** java.io.IOException;

**import** java.io.InputStreamReader;

**import** java.util.ArrayList;

**import** java.util.List;

**import** org.joml.Vector2f;

**import** org.joml.Vector3f;

/\*\*No longer needed as I do no longer plan on using Blender to generate objects<br>

\* **@see** <a href="https://en.wikipedia.org/wiki/Wavefront\_.obj\_file">https://en.wikipedia.org/wiki/Wavefront\_.obj\_file</a>

\*/

**public** **class** WavefrontMeshLoader {

/\*\*Loads a .obj file

\* **@param** fileName {@link WavefrontMeshLoader#readAllLines(String)}

\* **@return** A mesh object

\* **@throws** ClassNotFoundException

\* **@throws** IOException

\*/

**public** **static** Mesh loadMesh(String fileName) **throws** ClassNotFoundException, IOException {

List<String> lines = *readAllLines*(fileName);

List<Vector3f> vertices = **new** ArrayList<>();

List<Vector2f> textures = **new** ArrayList<>();

List<Vector3f> normals = **new** ArrayList<>();

List<Face> faces = **new** ArrayList<>();

**for** (String line : lines) {

String[] tokens = line.split("\\s+");

**switch** (tokens[0]) {

**case** "v":

// Geometric vertex

Vector3f vec3f = **new** Vector3f(

Float.*parseFloat*(tokens[1]),

Float.*parseFloat*(tokens[2]),

Float.*parseFloat*(tokens[3]));

vertices.add(vec3f);

**break**;

**case** "vt":

// Texture coordinate

Vector2f vec2f = **new** Vector2f(

Float.*parseFloat*(tokens[1]),

Float.*parseFloat*(tokens[2]));

textures.add(vec2f);

**break**;

**case** "vn":

// Vertex normal

Vector3f vec3fNorm = **new** Vector3f(

Float.*parseFloat*(tokens[1]),

Float.*parseFloat*(tokens[2]),

Float.*parseFloat*(tokens[3]));

normals.add(vec3fNorm);

**break**;

**case** "f":

Face face = **new** Face(tokens[1], tokens[2], tokens[3]);

faces.add(face);

**break**;

**default**:

// Ignore other lines

**break**;

}

}

**return** *reorderLists*(vertices, textures, normals, faces);

}

**private** **static** Mesh reorderLists(List<Vector3f> posList, List<Vector2f> textCoordList,

List<Vector3f> normList, List<Face> facesList) {

List<Integer> indices = **new** ArrayList<>();

// Create position array in the order it has been declared

**float**[] posArr = **new** **float**[posList.size() \* 3];

**int** i = 0;

**for** (Vector3f pos : posList) {

posArr[i \* 3] = pos.x;

posArr[i \* 3 + 1] = pos.y;

posArr[i \* 3 + 2] = pos.z;

i++;

}

**float**[] textCoordArr = **new** **float**[posList.size() \* 2];

**float**[] normArr = **new** **float**[posList.size() \* 3];

**for** (Face face : facesList) {

IdxGroup[] faceVertexIndices = face.getFaceVertexIndices();

**for** (IdxGroup indValue : faceVertexIndices) {

*processFaceVertex*(indValue, textCoordList, normList,

indices, textCoordArr, normArr);

}

}

**int**[] indicesArr = **new** **int**[indices.size()];

indicesArr = indices.stream().mapToInt((Integer v) -> v).toArray();

Mesh mesh = **new** Mesh(posArr, textCoordArr, normArr, indicesArr);

**return** mesh;

}

**private** **static** **void** processFaceVertex(IdxGroup indices, List<Vector2f> textCoordList,

List<Vector3f> normList, List<Integer> indicesList,

**float**[] texCoordArr, **float**[] normArr) {

// Set index for vertex coordinates

**int** posIndex = indices.idxPos;

indicesList.add(posIndex);

// Reorder texture coordinates

**if** (indices.idxTextCoord >= 0) {

Vector2f textCoord = textCoordList.get(indices.idxTextCoord);

texCoordArr[posIndex \* 2] = textCoord.x;

texCoordArr[posIndex \* 2 + 1] = 1 - textCoord.y;

}

**if** (indices.idxVecNormal >= 0) {

// Reorder vectornormals

Vector3f vecNorm = normList.get(indices.idxVecNormal);

normArr[posIndex \* 3] = vecNorm.x;

normArr[posIndex \* 3 + 1] = vecNorm.y;

normArr[posIndex \* 3 + 2] = vecNorm.z;

}

}

**protected** **static** **class** Face {

/\*\*

\* List of idxGroup groups for a face triangle (3 vertices per face).

\*/

**private** IdxGroup[] idxGroups = **new** IdxGroup[3];

**public** Face(String v1, String v2, String v3) {

idxGroups = **new** IdxGroup[3];

// Parse the lines

idxGroups[0] = parseLine(v1);

idxGroups[1] = parseLine(v2);

idxGroups[2] = parseLine(v3);

}

**private** IdxGroup parseLine(String line) {

IdxGroup idxGroup = **new** IdxGroup();

String[] lineTokens = line.split("/");

**int** length = lineTokens.length;

idxGroup.idxPos = Integer.*parseInt*(lineTokens[0]) - 1;

**if** (length > 1) {

// It can be empty if the obj does not define text coords

String textCoord = lineTokens[1];

idxGroup.idxTextCoord = textCoord.length() > 0 ? Integer.*parseInt*(textCoord) - 1 : IdxGroup.***NO\_VALUE***;

**if** (length > 2) {

idxGroup.idxVecNormal = Integer.*parseInt*(lineTokens[2]) - 1;

}

}

**return** idxGroup;

}

**public** IdxGroup[] getFaceVertexIndices() {

**return** idxGroups;

}

}

**protected** **static** **class** IdxGroup {

**public** **static** **final** **int** ***NO\_VALUE*** = -1;

**public** **int** idxPos;

**public** **int** idxTextCoord;

**public** **int** idxVecNormal;

**public** IdxGroup() {

idxPos = ***NO\_VALUE***;

idxTextCoord = ***NO\_VALUE***;

idxVecNormal = ***NO\_VALUE***;

}

}

/\*\*Reads all lines in a file. This is used for files that are stored internally within the .jar and not for files on the file system.

\* **@param** fileName name of the file

\* **@return** list of strings where each string is a line

\* **@throws** ClassNotFoundException

\* **@throws** IOException

\*/

**public** **static** List<String> readAllLines(String fileName) **throws** ClassNotFoundException, IOException {

List<String> list = **new** ArrayList<String>();

**try**(BufferedReader br = **new** BufferedReader(**new** InputStreamReader(Class.*forName*(WavefrontMeshLoader.**class**.getName()).getResourceAsStream(fileName)))) {

String line;

**while**((line = br.readLine()) != **null**)

list.add(line);

}

**return** list;

}

}

## Net.alevel.asteroids.engine.input

### Input.java

**package** net.alevel.asteroids.engine.input;

**import** java.util.BitSet;

**import** org.joml.Vector2d;

**import** org.joml.Vector2f;

**import** net.alevel.asteroids.engine.Window;

**import** net.alevel.asteroids.engine.input.enums.MouseBtns;

**import** net.alevel.asteroids.engine.input.enums.NonPrintableChars;

**import** net.alevel.asteroids.engine.input.enums.SpecialChars;

/\*\*Stores a sample of the keyboard and mouse input. It is updated every time {@link Input#input(Window)} is run

\*/

**public** **class** Input {

**private** **final** MouseInput mouse;

**private** **final** Vector2f deltaMouseMove; //create copies of mouse pos to store values at the instant they were recorded

**private** **final** Vector2d currentMousePos;

**private** **boolean** isMouseInWindow;

**private** **final** BitSet mouseBtnsPressed;

**private** **final** KeyBoardInput keyBoard; //keyboard does not update its variables until the update method is run so dont need to create copies

**public** Input() {

**this**.mouse = **new** MouseInput();

**this**.deltaMouseMove = **new** Vector2f();

**this**.currentMousePos = **new** Vector2d();

**this**.mouseBtnsPressed = **new** BitSet(**this**.mouse.getMouseButtonsPressed().size());

**this**.keyBoard = **new** KeyBoardInput();

}

/\*\*Used to initialise the internally used {@link Mouse} class

\* **@param** window The window to sample

\*/

**public** **void** init(Window window) {

**this**.mouse.init(window);

}

/\*\*Triggers the input to sample the mouse and keyboard input

\* **@param** window the window to sample

\*/

**public** **void** input(Window window) {

**this**.mouse.input(window);

**this**.deltaMouseMove.set(**this**.mouse.getDisplayVec());

**this**.currentMousePos.set(**this**.mouse.getCurrentVec());

**this**.isMouseInWindow = **this**.mouse.isMouseInWindow();

**this**.mouseBtnsPressed.clear();

**this**.mouseBtnsPressed.or(**this**.mouse.getMouseButtonsPressed()); //copy values not object reference, should be independant of each other

**this**.keyBoard.input(window.getWindowHandle());

}

/\*\*Queries the sample to check if the key specified was pressed

\* **@param** character the key to check

\* **@return** <strong>true</strong> if it was pressed, <strong>false</strong> if it wasn't

\*/

**public** **boolean** isKeyPressed(**char** character) {

**if**(character >= 65 && character <= 90) //if true, it is an alphabetical letter

**return** **this**.keyBoard.getAlphabetKeysPressed().get(character - 65);

**else** **if**(character >= 48 && character <= 57) //if true, it is a number

**return** **this**.keyBoard.getNumberKeysPressed().get(character - 48);

**else**

**throw** **new** IndexOutOfBoundsException("This method only accepts characters A-Z (capitals) and 0-9");

}

/\*\*Same as {@link Input#isKeyPressed(char)} but for special characters

\* **@param** character

\* **@return** <strong>true</strong> if it was pressed, <strong>false</strong> if it wasn't

\*/

**public** **boolean** isKeyPressed(SpecialChars character) {

**return** **this**.keyBoard.getSpecialCharacters().get(character.ordinal());

}

/\*\*Same as {@link Input#isKeyPressed(char)} but for non printable characters

\* **@param** character

\* **@return** <strong>true</strong> if it was pressed, <strong>false</strong> if it wasn't

\*/

**public** **boolean** isKeyPressed(NonPrintableChars character) {

**return** **this**.keyBoard.getNonPrintableCharacters().get(character.ordinal());

}

/\*\*Same as {@link Input#isKeyPressed(char)} but for mouse buttons

\* **@param** character

\* **@return** <strong>true</strong> if it was pressed, <strong>false</strong> if it wasn't

\*/

**public** **boolean** isMouseBtnPressed(MouseBtns btn) {

**return** **this**.mouseBtnsPressed.get(btn.ordinal());

}

/\*\*Gets the difference between the mouse position in the most recent sample and the mouse position in the sample just before that

\* **@return** 2D vector representing the movement of the mouse between samples

\*/

**public** Vector2f getDeltaMousePos() {

**return** **this**.deltaMouseMove;

}

/\*\*Gets the mouse position at the time of sampling

\* **@return** 2D vector representing that

\*/

**public** Vector2d getCurrentMousePos() {

**return** **this**.currentMousePos;

}

/\*\*Checks if the mouse was in the window at the time of sampling

\* **@return** <strong>true</strong> if it was pressed, <strong>false</strong> if it wasn't

\*/

**public** **boolean** isMouseInWindow() {

**return** **this**.isMouseInWindow;

}

}

### KeyBoardInput.java

**package** net.alevel.asteroids.engine.input;

**import** **static** org.lwjgl.glfw.GLFW.***GLFW\_PRESS***;

**import** **static** org.lwjgl.glfw.GLFW.*glfwGetKey*;

**import** java.util.BitSet;

**import** net.alevel.asteroids.engine.input.enums.NonPrintableChars;

**import** net.alevel.asteroids.engine.input.enums.SpecialChars;

/\*\*Stores a sample of all the keys pressed

\*/

**class** KeyBoardInput {

**private** **final** BitSet alphabetKeysPressed; //best way to store a set of flags

**private** **final** BitSet numberKeysPressed;

**private** **final** BitSet specialCharacters;

**private** **final** BitSet nonPrintableCharacters;

**public** KeyBoardInput() {

**this**.alphabetKeysPressed = **new** BitSet(26);

**this**.numberKeysPressed = **new** BitSet(10);

**this**.specialCharacters = **new** BitSet(SpecialChars.*values*().length);

**this**.nonPrintableCharacters = **new** BitSet(NonPrintableChars.*values*().length);

}

/\*\*Updates the keys pressed (takes new sample)

\* **@param** windowId ID of the window to listen to

\*/

**public** **void** input(**long** windowId) {

**for**(**int** i = 65; i <= 90; i++)

**this**.alphabetKeysPressed.set(i - 65, *glfwGetKey*(windowId, i) == ***GLFW\_PRESS***);

**for**(**int** i = 0; i <= 9; i++)

**this**.numberKeysPressed.set(i, *glfwGetKey*(windowId, i + 48) == ***GLFW\_PRESS***);

**for**(**int** i = 0; i < SpecialChars.*values*().length; i++)

**this**.specialCharacters.set(i, *glfwGetKey*(windowId, SpecialChars.*values*()[i].getGlId()) == ***GLFW\_PRESS***);

**for**(**int** i = 0; i < NonPrintableChars.*values*().length; i++)

**this**.nonPrintableCharacters.set(i, *glfwGetKey*(windowId, NonPrintableChars.*values*()[i].getGlId()) == ***GLFW\_PRESS***);

}

**public** BitSet getAlphabetKeysPressed() {

**return** alphabetKeysPressed;

}

**public** BitSet getNumberKeysPressed() {

**return** numberKeysPressed;

}

**public** BitSet getSpecialCharacters() {

**return** specialCharacters;

}

**public** BitSet getNonPrintableCharacters() {

**return** nonPrintableCharacters;

}

}

### MouseInput.java

**package** net.alevel.asteroids.engine.input;

**import** org.joml.Vector2d;

**import** org.joml.Vector2f;

**import** net.alevel.asteroids.engine.Window;

**import** **static** org.lwjgl.glfw.GLFW.\*;

**import** java.util.BitSet;

/\*\*Can get data on mouse activity with an instance of this class

\*/

**class** MouseInput {

**private** **final** Vector2d previousPos;

**private** **final** Vector2d currentPos;

**private** **final** Vector2f displVec; //the variables in these instances can change but the instances should not change.

**private** **boolean** inWindow; //true is cursor is hovering in the window, false if outside window

**private** **final** BitSet mouseButtonsPressed;

**public** MouseInput() {

**this**.previousPos = **new** Vector2d(-1, -1);

**this**.currentPos = **new** Vector2d(0, 0);

**this**.displVec = **new** Vector2f();

**this**.mouseButtonsPressed = **new** BitSet(3);

}

/\*\*Setup mouse event listeners

\*/

**public** **void** init(Window window) {

//glfwSetCursorPos(window.getWindowHandle(), window.getWidth() / 2, window.getHeight() / 2); //stops camera constantly moving glitch (it makes it worse) **TODO**

*glfwSetInputMode*(window.getWindowHandle(), ***GLFW\_CURSOR***, ***GLFW\_CURSOR\_HIDDEN***); //hides the cursor

*glfwSetCursorPosCallback*(window.getWindowHandle(), (windowHande, xpos, ypos) -> { //updates the currentPos values with the current cursor location

**this**.currentPos.x = xpos;

**this**.currentPos.y = ypos;

});

*glfwSetCursorEnterCallback*(window.getWindowHandle(), (windowHandle, enteredWindow) -> { //updates the inWindow boolean

**this**.inWindow = enteredWindow;

});

*glfwSetMouseButtonCallback*(window.getWindowHandle(), (windowHandle, button, action, mode) -> { //updates the Btn booleans so the logic can know if buttons were pressed on the mouse

**this**.mouseButtonsPressed.set(0, button == ***GLFW\_MOUSE\_BUTTON\_1*** && action == ***GLFW\_PRESS***);

**this**.mouseButtonsPressed.set(1, button == ***GLFW\_MOUSE\_BUTTON\_2*** && action == ***GLFW\_PRESS***);

**this**.mouseButtonsPressed.set(2, button == ***GLFW\_MOUSE\_BUTTON\_3*** && action == ***GLFW\_PRESS***);

});

}

**public** Vector2f getDisplayVec() {

**return** **this**.displVec;

}

**public** Vector2d getCurrentVec() {

**return** **this**.currentPos;

}

**public** **void** input(Window window) {

//this.displVec.x = 0;

//this.displVec.y = 0;

**if**(**this**.previousPos.x > 0 && **this**.previousPos.y > 0 && **this**.inWindow) {

**double** deltax = **this**.currentPos.x - **this**.previousPos.x;

**double** deltay = **this**.currentPos.y - **this**.previousPos.y;

**if**(deltax != 0)

**this**.displVec.y += (**float**) deltax; //by switching y and x, the look inverts

**if**(deltay != 0)

**this**.displVec.x += (**float**) deltay;

}

**this**.previousPos.x = **this**.currentPos.x;

**this**.previousPos.y = **this**.currentPos.y;

*glfwSetCursorPos*(window.getWindowHandle(), window.getWidth() / 2, window.getHeight() / 2); //sets the cursor to the centre of the screen so when doing large turns, cursor has enough space to move

}

**public** BitSet getMouseButtonsPressed() {

**return** **this**.mouseButtonsPressed;

}

**public** **boolean** isMouseInWindow() {

**return** **this**.inWindow;

}

}

## Net.alevel.asteroids.engine.input.enums

### MouseBtns.java

**package** net.alevel.asteroids.engine.input.enums;

/\*\*Enum for 3 of the common mouse buttons

\*/

**public** **enum** MouseBtns {

***LEFT\_CLICK***,

***RIGHT\_CLICK***,

***MIDDLE\_CLICK***;

}

### NonPrintableChars.java

**package** net.alevel.asteroids.engine.input.enums;

**import** **static** org.lwjgl.glfw.GLFW.***GLFW\_KEY\_BACKSPACE***;

**import** **static** org.lwjgl.glfw.GLFW.***GLFW\_KEY\_DELETE***;

**import** **static** org.lwjgl.glfw.GLFW.***GLFW\_KEY\_END***;

**import** **static** org.lwjgl.glfw.GLFW.***GLFW\_KEY\_ENTER***;

**import** **static** org.lwjgl.glfw.GLFW.***GLFW\_KEY\_ESCAPE***;

**import** **static** org.lwjgl.glfw.GLFW.***GLFW\_KEY\_HOME***;

**import** **static** org.lwjgl.glfw.GLFW.***GLFW\_KEY\_INSERT***;

**import** **static** org.lwjgl.glfw.GLFW.***GLFW\_KEY\_LEFT\_ALT***;

**import** **static** org.lwjgl.glfw.GLFW.***GLFW\_KEY\_LEFT\_CONTROL***;

**import** **static** org.lwjgl.glfw.GLFW.***GLFW\_KEY\_LEFT\_SHIFT***;

**import** **static** org.lwjgl.glfw.GLFW.***GLFW\_KEY\_PAGE\_DOWN***;

**import** **static** org.lwjgl.glfw.GLFW.***GLFW\_KEY\_PAGE\_UP***;

**import** **static** org.lwjgl.glfw.GLFW.***GLFW\_KEY\_RIGHT\_ALT***;

**import** **static** org.lwjgl.glfw.GLFW.***GLFW\_KEY\_RIGHT\_CONTROL***;

**import** **static** org.lwjgl.glfw.GLFW.***GLFW\_KEY\_RIGHT\_SHIFT***;

**import** **static** org.lwjgl.glfw.GLFW.***GLFW\_KEY\_TAB***;

/\*\*Represents keys that do not print characters<br>

\* E.g. Shift, Ctrl

\*/

**public** **enum** NonPrintableChars {

***BACKSPACE***(***GLFW\_KEY\_BACKSPACE***),

***ENTER***(***GLFW\_KEY\_ENTER***),

***RIGHT\_SHIFT***(***GLFW\_KEY\_RIGHT\_SHIFT***),

***RIGHT\_CTRL***(***GLFW\_KEY\_RIGHT\_CONTROL***),

***ALT\_GR***(***GLFW\_KEY\_RIGHT\_ALT***),

***ALT***(***GLFW\_KEY\_LEFT\_ALT***),

***LEFT\_CTRL***(***GLFW\_KEY\_LEFT\_CONTROL***),

***LEFT\_SHIFT***(***GLFW\_KEY\_LEFT\_SHIFT***),

***TAB***(***GLFW\_KEY\_TAB***),

***INSERT***(***GLFW\_KEY\_INSERT***),

***HOME***(***GLFW\_KEY\_HOME***),

***PAGE\_UP***(***GLFW\_KEY\_PAGE\_UP***),

***PAGE\_DOWN***(***GLFW\_KEY\_PAGE\_DOWN***),

***END***(***GLFW\_KEY\_END***),

***DELETE***(***GLFW\_KEY\_DELETE***),

***ESCAPE***(***GLFW\_KEY\_ESCAPE***);

**private** **final** **int** glId;

**private** NonPrintableChars(**int** glId) {

**this**.glId = glId;

}

**public** **int** getGlId() {

**return** **this**.glId;

}

}

### SpecialChars.java

**package** net.alevel.asteroids.engine.input.enums;

**import** **static** org.lwjgl.glfw.GLFW.***GLFW\_KEY\_APOSTROPHE***;

**import** **static** org.lwjgl.glfw.GLFW.***GLFW\_KEY\_BACKSLASH***;

**import** **static** org.lwjgl.glfw.GLFW.***GLFW\_KEY\_COMMA***;

**import** **static** org.lwjgl.glfw.GLFW.***GLFW\_KEY\_EQUAL***;

**import** **static** org.lwjgl.glfw.GLFW.***GLFW\_KEY\_GRAVE\_ACCENT***;

**import** **static** org.lwjgl.glfw.GLFW.***GLFW\_KEY\_LEFT\_BRACKET***;

**import** **static** org.lwjgl.glfw.GLFW.***GLFW\_KEY\_MINUS***;

**import** **static** org.lwjgl.glfw.GLFW.***GLFW\_KEY\_PERIOD***;

**import** **static** org.lwjgl.glfw.GLFW.***GLFW\_KEY\_RIGHT\_BRACKET***;

**import** **static** org.lwjgl.glfw.GLFW.***GLFW\_KEY\_SEMICOLON***;

**import** **static** org.lwjgl.glfw.GLFW.***GLFW\_KEY\_SLASH***;

**import** **static** org.lwjgl.glfw.GLFW.***GLFW\_KEY\_SPACE***;

/\*\*These are characters that are neither letters or numbers

\*/

**public** **enum** SpecialChars {

***MINUS***(***GLFW\_KEY\_MINUS***),

***EQUALS***(***GLFW\_KEY\_EQUAL***),

***OPEN\_SQUARE\_BRACKET***(***GLFW\_KEY\_LEFT\_BRACKET***),

***CLOSE\_SQUARE\_BRACKET***(***GLFW\_KEY\_RIGHT\_BRACKET***),

***SEMICOLON***(***GLFW\_KEY\_SEMICOLON***),

***APOSTROPHE***(***GLFW\_KEY\_APOSTROPHE***),

***COMMA***(***GLFW\_KEY\_COMMA***),

***FULL\_STOP***(***GLFW\_KEY\_PERIOD***),

***FORWARD\_SLASH***(***GLFW\_KEY\_SLASH***),

***BACK\_SLASH***(***GLFW\_KEY\_BACKSLASH***),

***GRAVE***(***GLFW\_KEY\_GRAVE\_ACCENT***),

***SPACE***(***GLFW\_KEY\_SPACE***);

**private** **final** **int** glId;

**private** SpecialChars(**int** glId) {

**this**.glId = glId;

}

**public** **int** getGlId() {

**return** **this**.glId;

}

}

## net.alevel.asteroids.engine.objects

### NonRenderableObject.java

**package** net.alevel.asteroids.engine.objects;

**import** org.joml.Vector3f;

**import** net.alevel.asteroids.engine.GameEngine;

**import** net.alevel.asteroids.game.objects.GameObjects;

/\*\*Base class for all object types. Simply holds values for position, rotation and scale (size)

\*/

**public** **abstract** **class** NonRenderableObject {

**protected** Vector3f position;

**protected** Vector3f rotation;

**protected** **float** scale;

**public** NonRenderableObject() {

**this**.position = **new** Vector3f();

**this**.rotation = **new** Vector3f();

}

**public** NonRenderableObject(Vector3f pos, Vector3f rot) {

**this**.position = pos;

**this**.rotation = rot;

}

/\*\*This function runs on every update in the simulation

\* **@param** time the in game time (number of milliseconds since simulation start)

\*/

**public** **void** update(**float** time) {

**this**.onUpdate(time);

//some code that should run in all types of objects

}

**protected** **abstract** **void** onUpdate(**float** time);

/\*\*This method exists so objects using opencl can clean up before opengl starts loading data. This is to make sure the GPU doesnt run out of memory

\* **@param** time the time interval (not accumulated time) between each update call (should be reciprocal of {@link GameEngine#TARGET\_UPS}

\*/

**public** **abstract** **void** onUpdateFinish(**float** time);

/\*\*Runs every time an instance of the class is spawned into the world

\* **@param** objectsManager

\*/

**public** **abstract** **void** onSpawn(GameObjects objectsManager);

/\*\*Runs every time an instance of the class is despawned from the world

\* **@param** objectsManager

\*/

**public** **abstract** **void** onDespawn(GameObjects objectsManager);

/\*\*Runs when the object is no longer needed and about to be destroyed.

\*/

**public** **abstract** **void** cleanUp();

@Override

**public** String toString() {

**return** "(pos -> " + **this**.position + " rot -> " + **this**.rotation + ")";

}

**public** NonRenderableObject setScale(**float** scale) {

**this**.scale = scale;

**return** **this**;

}

**public** NonRenderableObject enlarge(**float** dx) {

**this**.scale += dx;

**return** **this**;

}

**public** NonRenderableObject shrink(**float** dx) {

**return** **this**.enlarge(-dx);

}

/\*\*Note: this copies the values in the vector.

\* It does not reference the vector being passed (i.e. any changes made to the vector being passed will not affect the vector stored in this object)

\*/

**public** NonRenderableObject setPosition(Vector3f position) {

**this**.position.set(position);

**return** **this**;

}

**public** NonRenderableObject translate(Vector3f delta) {

**this**.position.add(delta);

**return** **this**;

}

**public** NonRenderableObject translate(**float** dx, **float** dy, **float** dz) {

**this**.position.add(dx, dy, dz);

**return** **this**;

}

**public** NonRenderableObject setPosition(**float** x, **float** y, **float** z) {

**this**.position.set(x, y, z);

**return** **this**;

}

/\*\*Note: this copies the values in the vector.

\* It does not reference the vector being passed (i.e. any changes made to the vector being passed will not affect the vector stored in this object)

\*/

**public** NonRenderableObject setRotation(Vector3f rotation) {

**this**.rotation.set(rotation);

**return** **this**;

}

**public** NonRenderableObject rotate(Vector3f delta) {

**this**.rotation.add(delta);

**return** **this**;

}

**public** NonRenderableObject rotate(**float** dx, **float** dy, **float** dz) {

**this**.rotation.add(dx, dy, dz);

**return** **this**;

}

**public** NonRenderableObject setRotation(**float** x, **float** y, **float** z) {

**this**.rotation.set(x, y, z);

**return** **this**;

}

**public** Vector3f getPosition() {

**return** **this**.position;

}

**public** **float** getScale() {

**return** **this**.scale;

}

**public** Vector3f getRotation() {

**return** **this**.rotation;

}

}

### GameObject.java

**package** net.alevel.asteroids.engine.objects;

**import** org.joml.Vector3f;

**import** net.alevel.asteroids.engine.graphics.Mesh;

/\*\*Represents a generic in game object that can be rendered.

\*/

**public** **abstract** **class** GameObject **extends** NonRenderableObject {

**protected** **final** Mesh mesh;

**public** GameObject(Mesh mesh) {

**this**.mesh = mesh;

**this**.position = **new** Vector3f();

**this**.scale = 1;

**this**.rotation = **new** Vector3f();

}

**public** Mesh getMesh() {

**return** **this**.mesh;

}

@Override

**public** **void** cleanUp() {

**this**.mesh.cleanUp();

}

}

## net.alevel.asteroids.engine.utils

### Pair.java

**package** net.alevel.asteroids.engine.utils;

/\*\*Generic class with 2 objects. The data types of both objects are specified at instantiation/declaration

\* **@param** <T1> data type of first object

\* **@param** <T2> data type of second object

\*/

**public** **class** Pair<T1, T2> {

**private** T1 o1;

**private** T2 o2;

**public** Pair(T1 o1, T2 o2) {

**this**.o1 = o1;

**this**.o2 = o2;

}

**public** T1 getO1() {

**return** **this**.o1;

}

**public** **void** setO1(T1 o1) {

**this**.o1 = o1;

}

**public** T2 getO2() {

**return** **this**.o2;

}

**public** **void** setO2(T2 o2) {

**this**.o2 = o2;

}

}

## net.alevel.asteroids.game.cl

### CLManager.java

**package** net.alevel.asteroids.game.cl;

**import** **static** org.jocl.CL.***CL\_CONTEXT\_PLATFORM***;

**import** **static** org.jocl.CL.\*;

**import** **static** org.jocl.CL.*clCreateCommandQueueWithProperties*;

**import** **static** org.jocl.CL.*clCreateContext*;

**import** **static** org.jocl.CL.*clGetDeviceIDs*;

**import** **static** org.jocl.CL.*clGetPlatformIDs*;

**import** **static** org.jocl.CL.*clReleaseCommandQueue*;

**import** **static** org.jocl.CL.*clReleaseContext*;

**import** org.jocl.CL;

**import** org.jocl.cl\_command\_queue;

**import** org.jocl.cl\_context;

**import** org.jocl.cl\_context\_properties;

**import** org.jocl.cl\_device\_id;

**import** org.jocl.cl\_platform\_id;

**import** org.jocl.cl\_queue\_properties;

/\*\*Manages OpenCL code. It creates the context (specifies the device etc.) and the instruction queue that the CL code will run on

\* In this case it will specify a GPU and create a queue for that.

\* Based on this ->

\* <a href="https://github.com/gpu/JOCLSamples/blob/master/src/main/java/org/jocl/samples/JOCLSample.java">https://github.com/gpu/JOCLSamples/blob/master/src/main/java/org/jocl/samples/JOCLSample.java</a>

\*/

**public** **class** CLManager {

**private** **static** cl\_context *context*;

**private** **static** cl\_queue\_properties *queueProperties*;

**private** **static** cl\_command\_queue *commandQueue*;

//private static int preferredGroupSize;

/\*\*

\* Creates a CL context for the kernals.<br>

\* NOTE: These kernals will run on the GPU

\*/

**public** **static** **void** init() {

// The platform, device type and device number

// that will be used

**final** **int** platformIndex = 0;

**final** **long** deviceType = ***CL\_DEVICE\_TYPE\_GPU***;

**final** **int** deviceIndex = 0;

**final** **int**[] err = **new** **int**[1];

// Enable exceptions and subsequently omit error checks in this sample

CL.*setExceptionsEnabled*(**true**);

// Obtain the number of platforms

**int** numPlatformsArray[] = **new** **int**[1];

*clGetPlatformIDs*(0, **null**, numPlatformsArray);

**int** numPlatforms = numPlatformsArray[0];

// Obtain a platform ID

cl\_platform\_id platforms[] = **new** cl\_platform\_id[numPlatforms];

*clGetPlatformIDs*(platforms.length, platforms, **null**);

cl\_platform\_id platform = platforms[platformIndex];

// Initialize the context properties

cl\_context\_properties contextProperties = **new** cl\_context\_properties();

contextProperties.addProperty(***CL\_CONTEXT\_PLATFORM***, platform);

// Obtain the number of devices for the platform

**int** numDevicesArray[] = **new** **int**[1];

*clGetDeviceIDs*(platform, deviceType, 0, **null**, numDevicesArray);

//System.out.println(Arrays.toString(numDevicesArray));

**int** numDevices = numDevicesArray[0];

// Obtain a device ID

cl\_device\_id devices[] = **new** cl\_device\_id[numDevices];

*clGetDeviceIDs*(platform, deviceType, numDevices, devices, **null**);

cl\_device\_id device = devices[deviceIndex];

//clGetDeviceInfo(device, CL\_KERNEL\_PREFERRED\_WORK\_GROUP\_SIZE\_MULTIPLE, param\_value\_size, param\_value, param\_value\_size\_ret) returns warp width (SIMD width) by default on GPU

// Create a context for the selected device

*context* = *clCreateContext*(contextProperties, 1, **new** cl\_device\_id[] { device }, **null**, **null**, err);

//System.out.println("Context err: " + err[0]);

// Create a command-queue for the selected device

*queueProperties* = **new** cl\_queue\_properties();

*commandQueue* = *clCreateCommandQueueWithProperties*(*context*, device, *queueProperties*, **null**);

CL.*clFinish*(*commandQueue*);

}

/\*\*To be run on simulation shut down. It shuts down the context and instruction queue

\*/

**public** **static** **void** cleanUp() {

*clReleaseCommandQueue*(*commandQueue*);

*clReleaseContext*(*context*);

}

**public** **static** cl\_context getContext() {

**return** *context*;

}

**public** **static** cl\_queue\_properties getQueueProperties() {

**return** *queueProperties*;

}

**public** **static** cl\_command\_queue getCommandQueue() {

**return** *commandQueue*;

}

}

### CLUtil.class

**package** net.alevel.asteroids.game.cl;

**import** **static** org.jocl.CL.*clBuildProgram*;

**import** **static** org.jocl.CL.*clCreateProgramWithSource*;

**import** java.io.BufferedReader;

**import** java.io.IOException;

**import** java.io.InputStreamReader;

**import** org.jocl.cl\_context;

**import** org.jocl.cl\_program;

/\*\*Contains static methods for general use within OpenCL related functions

\*/

**public** **class** CLUtil {

/\*\*Loads an openCL source code file (stored within the .jar)

\* **@param** path the path to the resource

\* **@param** classReference the class who's class loader will be used to load the resource

\* **@param** context The OpenCL context

\* **@return** An OpenCL program object

\* **@throws** IOException

\*/

**public** **static** cl\_program loadProgram(String path, Class<?> classReference, cl\_context context) **throws** IOException {

String programString = "";

**try**(BufferedReader br = **new** BufferedReader(**new** InputStreamReader(classReference.getResourceAsStream(path)))) {

String line;

**while**((line = br.readLine()) != **null**)

programString += line + "\n";

}

//System.out.println(programString);

cl\_program program = *clCreateProgramWithSource*(context,

1,

**new** String[] {programString},

**null**,

**null**);

*clBuildProgram*(program, 0, **null**, **null**, **null**, **null**);

**return** program;

}

}

## net.alevel.asteroids.game.noise

### ImprovedPerlin.java

**package** net.alevel.asteroids.game.noise;

/\*\*A faster version of {@link Perlin2}<br>

\* <i>unfinished</i>

\*/

**public** **class** ImprovedPerlin {

**static** **final** **int**[] ***permutation*** = { 151, 160, 137, 91, 90, 15, 131, 13, 201, 95, 96, 53, 194, 233, 7, 225, 140, 36,

103, 30, 69, 142, 8, 99, 37, 240, 21, 10, 23, 190, 6, 148, 247, 120, 234, 75, 0, 26, 197, 62, 94, 252, 219,

203, 117, 35, 11, 32, 57, 177, 33, 88, 237, 149, 56, 87, 174, 20, 125, 136, 171, 168, 68, 175, 74, 165, 71,

134, 139, 48, 27, 166, 77, 146, 158, 231, 83, 111, 229, 122, 60, 211, 133, 230, 220, 105, 92, 41, 55, 46,

245, 40, 244, 102, 143, 54, 65, 25, 63, 161, 1, 216, 80, 73, 209, 76, 132, 187, 208, 89, 18, 169, 200, 196,

135, 130, 116, 188, 159, 86, 164, 100, 109, 198, 173, 186, 3, 64, 52, 217, 226, 250, 124, 123, 5, 202, 38,

147, 118, 126, 255, 82, 85, 212, 207, 206, 59, 227, 47, 16, 58, 17, 182, 189, 28, 42, 223, 183, 170, 213,

119, 248, 152, 2, 44, 154, 163, 70, 221, 153, 101, 155, 167, 43, 172, 9, 129, 22, 39, 253, 19, 98, 108, 110,

79, 113, 224, 232, 178, 185, 112, 104, 218, 246, 97, 228, 251, 34, 242, 193, 238, 210, 144, 12, 191, 179,

162, 241, 81, 51, 145, 235, 249, 14, 239, 107, 49, 192, 214, 31, 181, 199, 106, 157, 184, 84, 204, 176, 115,

121, 50, 45, 127, 4, 150, 254, 138, 236, 205, 93, 222, 114, 67, 29, 24, 72, 243, 141, 128, 195, 78, 66, 215,

61, 156, 180 };

**private** **int** p[] = **new** **int**[512];

**public** ImprovedPerlin() {

**for** (**int** i = 0; i < 256; i++) {

p[256 + i] = ***permutation***[i];

p[i] = ***permutation***[i];

}

}

**public** **double** get(**double** x, **double** y) {

// find square that point is in

**int** x0 = (**int**) Math.*floor*(x) & 255;

**int** y0 = (**int**) Math.*floor*(y) & 255;

//find position of point within square

x -= Math.*floor*(x);

y -= Math.*floor*(y);

//get fade curves

**double** u = fade(x);

**double** v = fade(y);

**return** 0;

}

**private** **double** grad(**int** hash, **double** x, **double** y) {

**return** 0.0;

}

**private** **int** hash(**double** x, **double** y) {

**return** p[p[(**int**)x] + (**int**)y];

}

**private** **double** lerp(**double** a, **double** b, **double** x) {

**return** a + x \* (b - a);

}

**private** **double** fade(**double** x) {

**return** (6 \* Math.*pow*(x, 5)) - (15 \* Math.*pow*(x, 4)) + (10 \* Math.*pow*(x, 3));

}

}

### Perlin.java

**package** net.alevel.asteroids.game.noise;

**import** java.util.Random;

**import** org.joml.Vector2d;

/\*\*

\* **@deprecated** Use {@link Perlin2} as this has to have a predefined size during instantiation.

\* **@see** Perlin2

\*/

@Deprecated

**public** **class** ~~Perlin~~ {

**private** **static** **final** Vector2d[] ***gVectorList*** = {

**new** Vector2d(1, 0),

**new** Vector2d(1, 1),

**new** Vector2d(0, 1),

**new** Vector2d(-1, 1),

**new** Vector2d(-1, 0),

**new** Vector2d(-1, -1),

**new** Vector2d(0, -1),

**new** Vector2d(1, -1)

};

**private** **final** **int**[][] gVectors;

//private final int[][] influenceValues;

**public** Perlin(**int** width, **int** height, **long** seed) {

**this**.~~gVectors~~ = **new** **int**[width][height];

//this.influenceValues = new int[width][height];

Random rng = **new** Random(seed);

//System.out.println(width + " " + height);

**for**(**int** i = 0; i < width; i++) {

**for**(**int** j = 0; j < height; j++) {

**this**.~~gVectors~~[i][j] = rng.nextInt(8);

//this.influenceValues[i][j] = rng.nextInt(1);

//System.out.println(gVectors[i][j]);

}

}

}

**public** **double** get(**double** x, **double** y) {

//get reference to square that point is in. Where (x0, y0) represent bottom left corner

**int** x0 = (**int**) x,

x1 = x0 + 1,

y0 = (**int**) y,

y1 = y0 + 1;

//System.out.println(x + " " + y);

//calculate distance vectors

Vector2d[] distVectors = {

**new** Vector2d(x - x0, y - y0),

**new** Vector2d(x - x0, y - y1),

**new** Vector2d(x - x1, y - y0),

**new** Vector2d(x - x1, y - y1)

};

//System.out.println(Arrays.toString(distVectors));

**double**[] dotProducts = {

~~gVectorList~~[**this**.~~gVectors~~[x0][y0]].dot(distVectors[0]),

~~gVectorList~~[**this**.~~gVectors~~[x0][y1]].dot(distVectors[1]),

~~gVectorList~~[**this**.~~gVectors~~[x1][y0]].dot(distVectors[2]),

~~gVectorList~~[**this**.~~gVectors~~[x1][y1]].dot(distVectors[3]),

};

**double** u = ~~fade~~(x - x0),

v = ~~fade~~(y - y0);

**double** lerp1 = ~~lerp~~(dotProducts[0], dotProducts[1], v);

**double** lerp2 = ~~lerp~~(dotProducts[2], dotProducts[3], v);

**double** avg = ~~lerp~~(lerp1, lerp2, u);

//System.out.println(avg);

**return** avg;

}

**private** **double** lerp(**double** a, **double** b, **double** x) {

**return** a + x \* (b - a);

}

**private** **double** fade(**double** x) {

**return** (6 \* Math.*pow*(x, 5)) - (15 \* Math.*pow*(x, 4)) + (10 \* Math.*pow*(x, 3));

}

}

### Perlin2.java

**package** net.alevel.asteroids.game.noise;

**import** org.joml.Vector2d;

**import** net.alevel.asteroids.engine.graphics.Mesh;

**import** net.alevel.asteroids.game.objects.ModifiableMesh;

**import** net.alevel.asteroids.game.objects.StaticGameObject;

**import** net.alevel.asteroids.game.objects.shapes.Grid;

/\*\*Basic implementation of the 'Perlin noise' algorithm. It is based of this tutorial:

\* **@see** <a href="https://adrianb.io/2014/08/09/perlinnoise.html">https://adrianb.io/2014/08/09/perlinnoise.html</a>

\*/

**public** **class** Perlin2 {

**public** Perlin2() {

}

/\*\*Generates a height value for a specific 2D point on the height map

\* **@param** x

\* **@param** y

\* **@return** the height

\*/

**public** **double** get(**double** x, **double** y) {

// get reference to square that point is in. Where (x0, y0) represent bottom

// left corner

**int** x0 = (**int**) x, x1 = x0 + 1, y0 = (**int**) y, y1 = y0 + 1;

// System.out.println(x + " " + y);

// calculate distance vectors

Vector2d[] distVectors = { **new** Vector2d(x - x0, y - y0), **new** Vector2d(x - x0, y - y1),

**new** Vector2d(x - x1, y - y0), **new** Vector2d(x - x1, y - y1) };

// System.out.println(Arrays.toString(distVectors));

**double**[] dotProducts = {

getGradVec(x0, y0).dot(distVectors[0]),

getGradVec(x0, y1).dot(distVectors[1]),

getGradVec(x1, y0).dot(distVectors[2]),

getGradVec(x1, y1).dot(distVectors[3]),

};

**double** u = fade(x - x0), v = fade(y - y0);

**double** lerp1 = lerp(dotProducts[0], dotProducts[1], v);

**double** lerp2 = lerp(dotProducts[2], dotProducts[3], v);

**double** avg = lerp(lerp1, lerp2, u);

// System.out.println(avg);

**return** avg;

}

**private** Vector2d getGradVec(**int** x, **int** y) {

**return** ***gVectorList***[***p***[(***p***[x & 255] + y) & 255] & 7];

}

**private** **double** lerp(**double** a, **double** b, **double** x) {

**return** a + x \* (b - a);

}

**private** **double** fade(**double** x) {

**return** (6 \* Math.*pow*(x, 5)) - (15 \* Math.*pow*(x, 4)) + (10 \* Math.*pow*(x, 3));

}

**private** **static** **final** Vector2d[] ***gVectorList*** = {

**new** Vector2d(1, 0),

**new** Vector2d(1, 1),

**new** Vector2d(0, 1),

**new** Vector2d(-1, 1),

**new** Vector2d(-1, 0),

**new** Vector2d(-1, -1),

**new** Vector2d(0, -1),

**new** Vector2d(1, -1)

};

**private** **static** **final** **int**[] ***p*** = { 151, 160, 137, 91, 90, 15, 131, 13, 201, 95, 96, 53, 194, 233, 7, 225,

140, 36, 103, 30, 69, 142, 8, 99, 37, 240, 21, 10, 23, 190, 6, 148, 247, 120, 234, 75, 0, 26, 197, 62, 94,

252, 219, 203, 117, 35, 11, 32, 57, 177, 33, 88, 237, 149, 56, 87, 174, 20, 125, 136, 171, 168, 68, 175, 74,

165, 71, 134, 139, 48, 27, 166, 77, 146, 158, 231, 83, 111, 229, 122, 60, 211, 133, 230, 220, 105, 92, 41,

55, 46, 245, 40, 244, 102, 143, 54, 65, 25, 63, 161, 1, 216, 80, 73, 209, 76, 132, 187, 208, 89, 18, 169,

200, 196, 135, 130, 116, 188, 159, 86, 164, 100, 109, 198, 173, 186, 3, 64, 52, 217, 226, 250, 124, 123, 5,

202, 38, 147, 118, 126, 255, 82, 85, 212, 207, 206, 59, 227, 47, 16, 58, 17, 182, 189, 28, 42, 223, 183,

170, 213, 119, 248, 152, 2, 44, 154, 163, 70, 221, 153, 101, 155, 167, 43, 172, 9, 129, 22, 39, 253, 19, 98,

108, 110, 79, 113, 224, 232, 178, 185, 112, 104, 218, 246, 97, 228, 251, 34, 242, 193, 238, 210, 144, 12,

191, 179, 162, 241, 81, 51, 145, 235, 249, 14, 239, 107, 49, 192, 214, 31, 181, 199, 106, 157, 184, 84, 204,

176, 115, 121, 50, 45, 127, 4, 150, 254, 138, 236, 205, 93, 222, 114, 67, 29, 24, 72, 243, 141, 128, 195,

78, 66, 215, 61, 156, 180

};

/\*\*This function just creates an object used to test and demonstrate Perlin noise

\* **@return** a 2D perlin noise height map represented in 3D

\*/

**public** **static** StaticGameObject example() {

**int** width = 1000,

height = 1000;

ModifiableMesh grid = Grid.*create*(width, height, 1);

//System.out.println(grid.getPositions().length);

//System.out.println(grid.getIndices().length);

//Perlin noise = new Perlin(10, 10, new Random().nextLong());

Perlin2 noise = **new** Perlin2();

//Random rng = new Random();

//System.out.println(noise.get(0.05, 0.05));

**for**(**int** i = 0; i < width; i++) {

**for**(**int** j = 0; j < height; j++) {

**float** h = (**float**) (noise.get((**double**) i / 10, (**double**) j / 10) \* 10);

//System.out.println(((double)(i / 100)) + " " + ((double)(j / 100)) + " => " + height);

grid.changePosition((((i \* 500) + j) \* 3) + 1, h);

}

}

**return** **new** StaticGameObject(**new** Mesh(grid.getPositions(), grid.getPositions(), grid.getPositions(), grid.getIndices()));

}

}

## net.alevel.asteroids.game.objects

### Asteroid.java

**package** net.alevel.asteroids.game.objects;

**import** net.alevel.asteroids.engine.graphics.Mesh;

**import** net.alevel.asteroids.game.noise.Perlin2;

**import** net.alevel.asteroids.game.objects.shapes.MeshGen;

/\*\*Represents an asteroid.

\* The asteroid shapes are procedurally generated.

\*/

**public** **class** Asteroid **extends** StaticGameObject {

**public** Asteroid() {

**super**(*generate*());

**super**.scale = 10;

}

/\*\*Gets an icosphere from {@link MeshGen#modifiableSphere(float, int)} and wraps a perlin noise height map (generated using {@link Perlin2}) around it to generate an asteroid.

\* **@return**

\*/

**private** **static** Mesh generate() {

ModifiableMesh asteroid = MeshGen.*modifiableSphere*(5, 4);

Perlin2 noise2 = **new** Perlin2();

**for**(**int** i = 0; i < asteroid.getPositions().length / 100; i++) {

**for**(**int** j = 0; j < 100; j++) {

**float** height = (**float**) noise2.get((**double**) i / 500d, (**double**) j / 500d);

//System.out.println(height);

asteroid.changePosition((i \* 100) + j + 1, (**float**) (asteroid.getPositions()[(i \* 100) + j + 1] \* (height + 0.5)));

}

}

**return** **new** Mesh(asteroid.getPositions(), asteroid.getPositions(), asteroid.getPositions(), asteroid.getIndices());

}

}

### StaticGameObject.java

**package** net.alevel.asteroids.game.objects;

**import** net.alevel.asteroids.engine.graphics.Mesh;

**import** net.alevel.asteroids.engine.objects.GameObject;

/\*\*Represents a simple object that does not do anything on a simulation update

\*/

**public** **class** StaticGameObject **extends** GameObject {

**public** StaticGameObject(Mesh mesh) {

**super**(mesh);

}

@Override

**public** **void** onUpdate(**float** time) { //static objects do nothing

}

@Override

**public** **void** onUpdateFinish(**float** time) {

}

@Override

**public** **void** onSpawn(GameObjects objectsManager) {

// **TODO** Auto-generated method stub

}

@Override

**public** **void** onDespawn(GameObjects objectsManager) {

// **TODO** Auto-generated method stub

}

}

### Ship.java

**package** net.alevel.asteroids.game.objects;

**import** org.joml.Vector2f;

**import** org.joml.Vector3f;

**import** net.alevel.asteroids.engine.objects.GameObject;

**import** net.alevel.asteroids.game.objects.shapes.MeshGen;

/\*\*A standard ship

\*/

**public** **class** Ship **extends** ObjectAssembly {

**public** Ship() {

GameObject cockpit = **new** StaticGameObject(MeshGen.*triangularPrism*(**new** Vector2f(0, 0), **new** Vector2f(0, 1), **new** Vector2f(1, 0), 2));

GameObject leftArm = **new** StaticGameObject(MeshGen.*cube*(2, .5f, .5f));

GameObject rightArm = **new** StaticGameObject(MeshGen.*cube*(2, .5f, .5f));

**super**.addObject(cockpit, **new** Vector3f(-.5f, 0, 0));

**super**.addObject(rightArm, **new** Vector3f(0, 0, 1));

**super**.addObject(leftArm, **new** Vector3f(0, 0, -1));

}

}

### ObjectAssembly.java

**package** net.alevel.asteroids.game.objects;

**import** java.util.ArrayList;

**import** java.util.Collections;

**import** java.util.List;

**import** org.joml.Matrix3f;

**import** org.joml.Vector3f;

**import** net.alevel.asteroids.engine.objects.GameObject;

**import** net.alevel.asteroids.engine.objects.NonRenderableObject;

/\*\*Represents an Object made of multiple objects. These would be used to create more complex shapes (like ships)

\*/

**public** **class** ObjectAssembly **extends** NonRenderableObject {

**private** **final** List<GameObject> objects;

**private** **final** List<Vector3f> relativePositions; //The vector linking the centre of a single component to the centre of the whole 'object'

**public** ObjectAssembly() {

**this**(**new** Vector3f(), **new** Vector3f());

}

**public** ObjectAssembly(Vector3f position, Vector3f rotation) {

**super**(position, rotation);

**this**.objects = **new** ArrayList<GameObject>();

**this**.relativePositions = **new** ArrayList<Vector3f>();

}

@Override

**protected** **void** onUpdate(**float** time) {

**this**.rotateObjects();

**this**.moveObjects();

}

**private** **void** rotateObjects() {

Matrix3f rotation = **new** Matrix3f();

rotation.rotateX((**float**) Math.*toRadians*(**this**.rotation.x));

rotation.rotateY((**float**) Math.*toRadians*(**this**.rotation.y));

rotation.rotateZ((**float**) Math.*toRadians*(**this**.rotation.z));

**for**(**int** i = 0; i < **this**.objects.size(); i++) {

Vector3f newPos = **new** Vector3f(**this**.relativePositions.get(i));

newPos.mul(rotation);

newPos.add(**this**.position);

**this**.objects.get(i).setPosition(newPos);

**this**.objects.get(i).setRotation(-**this**.rotation.x, -**this**.rotation.y, -**this**.rotation.z);

}

}

**private** **void** moveObjects() {

**for**(**int** i = 0; i < **this**.objects.size(); i++)

**this**.objects.get(i).setPosition(**this**.objects.get(i).getPosition().add(**this**.position));

}

/\*\*Add new object to assembly

\* **@param** o the object

\* **@param** relativePos the position of this object relative to the centre of the whole assembly

\*/

**public** **void** addObject(GameObject o, Vector3f relativePos) {

**this**.objects.add(o);

o.setPosition(relativePos.add(**this**.position));

**this**.relativePositions.add(relativePos);

}

**public** List<GameObject> getObjects() {

**return** Collections.*unmodifiableList*(**this**.objects);

}

@Override

**public** **void** onUpdateFinish(**float** time) {

}

@Override

**public** **void** onSpawn(GameObjects objectsManager) {

**for**(GameObject i : **this**.objects)

objectsManager.spawnObject(i);

}

@Override

**public** **void** onDespawn(GameObjects objectsManager) {

**for**(GameObject i : **this**.objects)

objectsManager.despawnObject(i);

}

@Override

**public** **void** cleanUp() {

**for**(GameObject i : **this**.objects)

i.cleanUp();

}

}

### ModifiableMesh.java

**package** net.alevel.asteroids.game.objects;

**import** net.alevel.asteroids.engine.graphics.Mesh;

/\*\*Similar to {@link Mesh}. It stores positions, textures, normals and indices but does not load them into GL buffers as they could still change.<br>

\* {@link Mesh} contains non modifiable attributes due to the fact that it represents data already sent to the GPU

\*/

**public** **class** ModifiableMesh {

**private** **float**[] positions;

**private** **float**[] textures;

**private** **float**[] normals;

**private** **int**[] indices;

**public** ModifiableMesh(**float**[] positions, **int**[] indices) {

**this**.positions = positions;

**this**.indices = indices;

}

**public** **float**[] getPositions() {

**return** **this**.positions;

}

**public** **void** setPositions(**float**[] positions) {

**this**.positions = positions;

}

**public** **void** changePosition(**int** index, **float** value) {

**this**.positions[index] = value;

}

**public** **float**[] getTextures() {

**return** **this**.textures;

}

**public** **void** setTextures(**float**[] textures) {

**this**.textures = textures;

}

**public** **float**[] getNormals() {

**return** **this**.normals;

}

**public** **void** setNormals(**float**[] normals) {

**this**.normals = normals;

}

**public** **int**[] getIndices() {

**return** **this**.indices;

}

**public** **void** setIndices(**int**[] indices) {

**this**.indices = indices;

}

/\*\*Generates a non modifiable mesh using the attributes of 'this'.

\* **@return**

\*/

**public** Mesh generateMesh() {

**return** **new** Mesh(positions, textures, normals, indices);

}

}

### GameObjects.java

**package** net.alevel.asteroids.game.objects;

**import** java.util.ArrayList;

**import** java.util.Collections;

**import** java.util.List;

**import** net.alevel.asteroids.engine.objects.GameObject;

**import** net.alevel.asteroids.engine.objects.NonRenderableObject;

**import** net.alevel.asteroids.game.physics.RigidObject;

/\*\*Manages all objects in the world and puts them into separate lists so I can for example send all the renderable objects to the GPU with a simple command

\*/

**public** **class** GameObjects {

**private** **final** List<NonRenderableObject> allObjects;

**private** **final** List<GameObject> renderableObjects;

**private** **final** List<RigidObject> rigidObjects;

**public** GameObjects() {

**this**.allObjects = **new** ArrayList<NonRenderableObject>();

**this**.renderableObjects = **new** ArrayList<GameObject>();

**this**.rigidObjects = **new** ArrayList<RigidObject>();

}

**public** **void** spawnObject(NonRenderableObject o) {

o.onSpawn(**this**);

**this**.allObjects.add(o);

**if**(o **instanceof** GameObject) {

**this**.renderableObjects.add((GameObject) o);

**if**(o **instanceof** RigidObject)

**this**.rigidObjects.add((RigidObject) o);

}

}

**public** **void** despawnObject(NonRenderableObject o) {

o.onDespawn(**this**);

**this**.allObjects.remove(o);

**if**(o **instanceof** GameObject) {

**this**.renderableObjects.remove((GameObject) o);

**if**(o **instanceof** RigidObject)

**this**.rigidObjects.remove((RigidObject) o);

}

}

**public** **void** spawnAll(List<NonRenderableObject> os) {

**for**(**int** i = 0; i < os.size(); i++)

**this**.spawnObject(os.get(i));

}

**public** **void** spawnAll(NonRenderableObject[] os) {

**for**(**int** i = 0; i < os.length; i++)

**this**.spawnObject(os[i]);

}

**public** List<RigidObject> getRigidObjects(){

**return** Collections.*unmodifiableList*(**this**.rigidObjects);

}

**public** List<NonRenderableObject> getAllObjects(){

**return** Collections.*unmodifiableList*(**this**.allObjects);

}

**public** List<GameObject> getRenderableObjects(){

**return** Collections.*unmodifiableList*(**this**.renderableObjects);

}

**public** **int** totalObjectCount() {

**return** **this**.allObjects.size();

}

}

## net.alevel.asteroids.game.objects.shapes

### MeshGen.java

**package** net.alevel.asteroids.game.objects.shapes;

**import** java.util.ArrayList;

**import** java.util.Arrays;

**import** java.util.List;

**import** org.joml.Matrix2f;

**import** org.joml.Vector2f;

**import** org.joml.Vector3f;

**import** net.alevel.asteroids.engine.graphics.Mesh;

**import** net.alevel.asteroids.engine.objects.GameObject;

**import** net.alevel.asteroids.game.objects.ModifiableMesh;

/\*\*Creates different shapes.<br>

\* NOTE: when used in a {@link GameObject}, the position attribute refers to the position of the centre of the shape

\*/

**public** **class** MeshGen {

/\*\*Creates a mesh of a cube

\* **@param** length the cube length

\* **@param** height the cube height

\* **@param** width the cube width

\* **@return** a mesh of the cube

\*/

**public** **static** Mesh cube(**float** length, **float** height, **float** width) { //x, y, z

**float**[] positions = {

length / 2, height / 2, width / 2, //0

-length / 2, height / 2, width / 2, //1

length / 2, -height / 2, width / 2, //2

length / 2, height / 2, -width / 2, //3

-length / 2, -height / 2, width / 2, //4

length / 2, -height / 2, -width / 2, //5

-length / 2, height / 2, -width / 2, //6

-length / 2, -height / 2, -width / 2 //7

};

**int**[] indices = {

0, 1, 2,

1, 2, 4,

1, 6, 4,

7, 6, 4,

6, 7, 5,

7, 3, 5,

0, 2, 3,

3, 2, 5,

0, 1, 6,

0, 6, 3,

2, 4, 7,

2, 5, 7

};

**return** **new** Mesh(positions, positions, positions, indices);

}

/\*\*Constructs a triangle from the 3 vertices and extrudes it a certain length

\* **@param** v1 Vertex 1 of the triangle

\* **@param** v2 Vertex 2 of the triangle

\* **@param** v3 Vertex 3 of the triangle

\* **@param** length the length of the prism

\* **@return** a mesh of a triangular prism

\*/

**public** **static** Mesh triangularPrism(Vector2f v1, Vector2f v2, Vector2f v3, **float** length) {

Vector2f midpoint = **new** Vector2f(

(v1.x + v2.x + v3.x) / 3,

(v1.y + v2.y + v3.y) / 3);

Vector2f newV1 = v1.sub(midpoint);

Vector2f newV2 = v2.sub(midpoint);

Vector2f newV3 = v3.sub(midpoint);

**float**[] positions = {

newV1.x, newV1.y, length / 2, //0

newV2.x, newV2.y, length / 2, //1

newV3.x, newV3.y, length / 2, //2

newV1.x, newV1.y, -length / 2, //3

newV2.x, newV2.y, -length / 2, //4

newV3.x, newV3.y, -length / 2, //5

};

**int**[] indices = {

0, 1, 2,

3, 4, 5,

0, 3, 4,

0, 1, 4,

0, 5, 2,

0, 5, 3,

1, 4, 2,

2, 4, 5

};

**return** **new** Mesh(positions, positions, positions, indices);

}

**public** **static** Mesh sphere(**float** f) {

**return** *sphere*(f, 5);

}

/\*\*Creates a mesh of an icosphere

\* **@param** radius

\* **@param** resolution

\* **@return** a mesh of the sphere

\*/

**public** **static** Mesh sphere(**float** radius, **int** resolution) {

/\*GenOctahedron genOctahedron = new GenOctahedron(resolution);

List<Vector3f> positions = genOctahedron.getPositions();

float[] floats = new float[positions.size() \* 3];

for(int i = 0; i < positions.size(); i++) {

Vector3f vertex = positions.get(i);

float scale = radius - vertex.length();

float dX = vertex.x \* (scale / vertex.length());

float dY = vertex.y \* (scale / vertex.length());

float dZ = vertex.z \* (scale / vertex.length());

floats[i \* 3] = vertex.x + dX;

floats[i \* 3 + 1] = vertex.y + dY;

floats[i \* 3 + 2] = vertex.z + dZ;

}

return new Mesh(floats, floats, floats, genOctahedron.getIndices());\*/

ModifiableMesh m = *modifiableSphere*(radius, resolution);

**return** **new** Mesh(m.getPositions(), **null**, **null**, m.getIndices());

}

/\*\*Creates a sphere as a modifiable shape (used for asteroid generation)

\* **@param** radius

\* **@param** resolution the number of vertices on the sphere. The more there are, the more spherical it looks

\* **@return**

\*/

**public** **static** ModifiableMesh modifiableSphere(**float** radius, **int** resolution) {

GenOctahedron genOctahedron = **new** GenOctahedron(resolution);

List<Vector3f> positions = genOctahedron.getPositions();

**float**[] floats = **new** **float**[positions.size() \* 3];

**for**(**int** i = 0; i < positions.size(); i++) {

Vector3f vertex = positions.get(i);

**float** scale = radius - vertex.length();

**float** dX = vertex.x \* (scale / vertex.length());

**float** dY = vertex.y \* (scale / vertex.length());

**float** dZ = vertex.z \* (scale / vertex.length());

floats[i \* 3] = vertex.x + dX;

floats[i \* 3 + 1] = vertex.y + dY;

floats[i \* 3 + 2] = vertex.z + dZ;

}

**return** **new** ModifiableMesh(floats, genOctahedron.getIndices());

}

/\*\*Generates a cylinder using a default circle resolution of 4

\* **@param** radius

\* **@param** length

\* **@return**

\*/

**public** **static** Mesh cylinder(**float** radius, **float** length) {

**return** *cylinder*(radius, length, 4);

}

/\*\*Generates a cylinder

\* **@param** radius

\* **@param** length

\* **@param** circleResolution the number of vertices for the circle

\* **@return**

\*/

**public** **static** Mesh cylinder(**float** radius, **float** length, **int** circleResolution) {

Matrix2f rotationMatrix = **new** Matrix2f().rotate((**float**) (Math.***PI*** / 2) / circleResolution); //Where the angle is 90 degrees / circleResolution

List<Vector2f> pointsOnCircle = **new** ArrayList<Vector2f>(); //holds all points on a circle

Vector2f temp = **new** Vector2f(1, 0);

pointsOnCircle.add(temp);

**for**(**int** i = 0; i < 4 \* circleResolution; i++) { //to rotate around 360 degrees, do number of points per 90 degrees (circleResolution) 4 times

temp = **new** Vector2f().set(temp).mul(rotationMatrix);

pointsOnCircle.add(temp);

}

**float**[] positions = **new** **float**[pointsOnCircle.size() \* 6 + 6]; //\* 2 (because 2 circles) \* 3 (because each vertex has 3 components) therefore \* 6 + 6 for 2 centres

**for**(**int** i = 0; i < pointsOnCircle.size(); i++) {

positions[i \* 6] = pointsOnCircle.get(i).x;

positions[(i \* 6) + 1] = pointsOnCircle.get(i).y;

positions[(i \* 6) + 2] = -length / 2;

positions[(i \* 6) + 3] = pointsOnCircle.get(i).x;

positions[(i \* 6) + 4] = pointsOnCircle.get(i).y;

positions[(i \* 6) + 5] = length / 2;

}

//add centres

positions[positions.length - 6] = 0;

positions[positions.length - 5] = 0;

positions[positions.length - 4] = -length / 2;

positions[positions.length - 3] = 0;

positions[positions.length - 2] = 0;

positions[positions.length - 1] = length / 2;

**int**[] indices = **new** **int**[positions.length + (pointsOnCircle.size() \* 6)];

**int** i = 0;

**for**(**int** j = 0; i < pointsOnCircle.size() \* 6; i += 3, j++) {

indices[i] = j;

indices[i + 1] = (j + 1) % (pointsOnCircle.size() \* 2);

indices[i + 2] = (j + 2) % (pointsOnCircle.size() \* 2);

}

**for**(**int** j = 0, c = 0; c < pointsOnCircle.size(); i += 3, j += 2, c++) {

indices[i] = j;

indices[i + 1] = (positions.length / 3) - 2;

indices[i + 2] = j + 2;

}

**for**(**int** j = 1; i < indices.length - 2; i += 3, j += 2) {

indices[i] = j;

indices[i + 1] = (positions.length / 3) - 1;

indices[i + 2] = j + 2;

}

System.***out***.println(Arrays.*toString*(indices));

System.***out***.println(Arrays.*toString*(positions) + pointsOnCircle.size());

**return** **new** Mesh(positions, positions, positions, indices);

}

}

### Grid.java

**package** net.alevel.asteroids.game.objects.shapes;

**import** net.alevel.asteroids.game.objects.ModifiableMesh;

/\*\*Creates a 2D grid in 3D space

\*/

**public** **class** Grid {

**public** **static** ModifiableMesh create(**int** width, **int** length, **int** div) {

**float**[] points = **new** **float**[width \* length \* (1/div) \* 3];

//System.out.println(points.length);

//System.out.println(width + " " + length);

**int**[] indices = **new** **int**[(width - 1) \* (length - 1) \* 12];

**int** c = 0;

**for**(**float** i = 0; i < length; i += div) {

**for**(**float** j = 0; j < width; j += div) {

points[c] = i;

points[c + 1] = 0;

points[c + 2] = j;

c += 3;

//System.out.println(i + "f, 0f, " + j + "f, ");

}

}

//System.out.println(Arrays.toString(points));

c = 0;

**int** c2 = 0;

**for**(**int** i = 0; i < length; i++) {

**if**(i + 1 != length)

**for**(**int** j = 0; j < width; j++) {

**if**(j + 1 != width) {

indices[c2] = c;

indices[c2 + 1] = c + 1;

indices[c2 + 2] = c + width + 1;

indices[c2 + 3] = c;

indices[c2 + 4] = c + width;

indices[c2 + 5] = c + width + 1;

//System.out.println(c + ", " + (c + 1) + ", " + (c + width + 1) + ",");

//System.out.println(c + ", " + (c + width) + ", " + (c + width + 1) + ",");

}

c++;

c2 += 6;

}

}

//System.out.println(Arrays.toString(indices));

**return** **new** ModifiableMesh(points, indices);

}

/\*public static void debug(GameObjects objects) {

float[] points = {

/\*0.0f, 0, 0.0f,

0.0f, 0, 1.0f,

0.0f, 0, 2.0f,

0.0f, 0, 3.0f,

1.0f, 0, 0.0f,

1.0f, 0, 1.0f,

1.0f, 0, 2.0f,

1.0f, 0, 3.0f,

2.0f, 0, 0.0f,

2.0f, 0, 1.0f,

2.0f, 0, 2.0f,

2.0f, 0, 3.0f,

3.0f, 0, 0.0f,

3.0f, 0, 1.0f,

3.0f, 0, 2.0f,

3.0f, 0, 3.0f,

0.0f, 0f, 0.0f,

0.0f, 0f, 1.0f,

0.0f, 0f, 2.0f,

0.0f, 0f, 3.0f,

1.0f, 0f, 0.0f,

1.0f, 0f, 1.0f,

1.0f, 0f, 2.0f,

1.0f, 0f, 3.0f,

2.0f, 0f, 0.0f,

2.0f, 0f, 1.0f,

2.0f, 0f, 2.0f,

2.0f, 0f, 3.0f,

3.0f, 0f, 0.0f,

3.0f, 0f, 1.0f,

3.0f, 0f, 2.0f,

3.0f, 0f, 3.0f,

};

int[] indices = {

0, 1, 5,

0, 4, 5,

1, 2, 6,

1, 5, 6,

2, 3, 7,

2, 6, 7,

4, 5, 9,

4, 8, 9,

5, 6, 10,

5, 9, 10,

6, 7, 11,

6, 10, 11,

8, 9, 13,

8, 12, 13,

9, 10, 14,

9, 13, 14,

10, 11, 15,

10, 14, 15,

/\*0, 1, 5,

0, 4, 5,

1, 2, 6,

1, 5, 6,

2, 3, 7,

2, 6, 7,

4, 5, 9,

4, 8, 9,

5, 6, 10,

5, 9, 10,

6, 7, 11,

6, 10, 11,

8, 9, 13,

8, 12, 13,

9, 10, 14,

9, 13, 14,

10, 11, 15,

10, 14, 15,

//11, 12, 16,

//11, 15, 16,

/\*0, 4, 5,

0, 1, 5,

1, 5, 6,

2, 6, 7,

3, 7, 8,

4, 8, 9,

};

objects.spawnObject(new StaticGameObject(new Mesh(points, points, points, indices)));

}\*/

}

### GenOctahedron.java

**package** net.alevel.asteroids.game.objects.shapes;

**import** java.util.ArrayList;

**import** java.util.Arrays;

**import** java.util.List;

**import** org.joml.Vector3f;

/\*\*Generates an octahedron mesh using a specified number of triangles (resolution). Used for a variety of shapes like spheres.

\*/

**class** GenOctahedron {

**private** List<Vector3f> positions;

**private** **int**[] indices;

**public** GenOctahedron(**int** resolution) {

**this**.generate(resolution);

}

/\*\*Generates an octahedron

\* **@param** resolution

\*/

**public** **void** generate(**int** resolution) {

List<Vector3f> vertices = **new** ArrayList<Vector3f>();

vertices.add(**new** Vector3f(0, 0, 1));

vertices.add(**new** Vector3f(0, 0, -1));

vertices.add(**new** Vector3f(0, 1, 0));

vertices.add(**new** Vector3f(0, -1, 0));

vertices.add(**new** Vector3f(1, 0, 0));

vertices.add(**new** Vector3f(-1, 0, 0)); //indices 0 -> 5

List<**int**[]> indices = **new** ArrayList<**int**[]>();

indices.add(**new** **int**[] {0, 2, 4});

indices.add(**new** **int**[] {0, 3, 4});

indices.add(**new** **int**[] {1, 3, 5});

indices.add(**new** **int**[] {1, 5, 2});

indices.add(**new** **int**[] {2, 1, 4});

indices.add(**new** **int**[] {3, 1, 4});

indices.add(**new** **int**[] {5, 0, 2});

indices.add(**new** **int**[] {5, 0, 3});

**for**(**int** i = 0; i < resolution; i++)

*splitTriangles*(vertices, indices);

**this**.positions = vertices;

**this**.indices = **new** **int**[indices.size() \* 3];

**for**(**int** i = 0; i < indices.size(); i++) {

**int**[] triangle = indices.get(i);

**this**.indices[i \* 3] = triangle[0];

**this**.indices[i \* 3 + 1] = triangle[1];

**this**.indices[i \* 3 + 2] = triangle[2];

}

//System.out.println(Arrays.toString(floats));

//System.out.println(Arrays.toString(ints));

//System.out.println(floats.length + " <- number of floats in sphere");

//return new Mesh(floats, floats, floats, ints);

}

**private** **static** **void** getMidpointTriangle(**int**[] selectedVertices, List<Vector3f> vertices, List<**int**[]> indices) {

Vector3f[] midpoints = **new** Vector3f[3];

midpoints[0] = *getMidpoint*(vertices.get(selectedVertices[0]), vertices.get(selectedVertices[1]));

midpoints[1] = *getMidpoint*(vertices.get(selectedVertices[1]), vertices.get(selectedVertices[2]));

midpoints[2] = *getMidpoint*(vertices.get(selectedVertices[2]), vertices.get(selectedVertices[0]));

**int** firstIndex = vertices.size();

vertices.addAll(Arrays.*asList*(midpoints));

List<**int**[]> newTriangles = **new** ArrayList<**int**[]>();

newTriangles.add(**new** **int**[] {firstIndex, firstIndex + 1, firstIndex + 2});

newTriangles.add(**new** **int**[] {firstIndex, selectedVertices[0], firstIndex + 2});

newTriangles.add(**new** **int**[] {firstIndex, selectedVertices[1], firstIndex + 1});

newTriangles.add(**new** **int**[] {firstIndex + 1, selectedVertices[2], firstIndex + 2});

indices.remove(selectedVertices);

indices.addAll(newTriangles);

}

**private** **static** **void** splitTriangles(List<Vector3f> vertices, List<**int**[]> indices) {

@SuppressWarnings("unchecked")

ArrayList<**int**[]> indicesCopy = (ArrayList<**int**[]>) ((ArrayList<**int**[]>) indices).clone();

//System.out.println(indicesCopy);

**for**(**int**[] i : indicesCopy)

*getMidpointTriangle*(i, vertices, indices);

}

**private** **static** Vector3f getMidpoint(Vector3f a, Vector3f b) {

**return** **new** Vector3f(a).add(b).div(2);

}

**public** List<Vector3f> getPositions() {

**return** **this**.positions;

}

/\*\*Same as {@link GenOctahedron#getPositions()} but as an array of floats rather than a 3D Vector list

\* **@return**

\*/

**public** **float**[] getPositionsAsFloats() {

**float**[] floats = **new** **float**[**this**.positions.size() \* 3];

**for**(**int** i = 0; i < **this**.positions.size(); i++) {

floats[i \* 3] = **this**.positions.get(i).x;

floats[i \* 3 + 1] = **this**.positions.get(i).y;

floats[i \* 3 + 2] = **this**.positions.get(i).z;

}

**return** floats;

}

/\*\*Get the indices used to specify which vertices are used for which triangle

\* **@return**

\*/

**public** **int**[] getIndices() {

**return** **this**.indices;

}

}

## net.alevel.asteroids.game.physics

### RigidObject.java

**package** net.alevel.asteroids.game.physics;

**import** **static** org.jocl.CL.***CL\_TRUE***;

**import** **static** org.jocl.CL.*clEnqueueReadBuffer*;

**import** **static** org.jocl.CL.*clFinish*;

**import** **static** org.jocl.CL.*clReleaseMemObject*;

**import** org.jocl.Pointer;

**import** org.jocl.Sizeof;

**import** org.jocl.cl\_mem;

**import** org.joml.Vector3f;

**import** net.alevel.asteroids.engine.graphics.Mesh;

**import** net.alevel.asteroids.engine.objects.GameObject;

**import** net.alevel.asteroids.game.cl.CLManager;

**import** net.alevel.asteroids.game.objects.GameObjects;

/\*\*Represents an in game object that should be effected by physics systems (i.e. have a hitbox)

\*/

**public** **class** RigidObject **extends** GameObject {

**private** cl\_mem worldVertices; //sub buffer of the worldVertices buffer

**private** **float**[] worldVerticesArr;

**public** RigidObject(Mesh mesh) {

**super**(mesh);

**this**.worldVerticesArr = **new** **float**[mesh.getVertices().length];

}

/\*\*Runs every time this object collides with another.<br>

\* Currently this method just turns the object red. For more advanced functionality, this method should be overridden.

\* **@param** otherObject

\*/

**public** **void** onCollision(RigidObject otherObject) {

**super**.getMesh().setColour(**new** Vector3f(1, 0, 0));

//GameLogic.getInstance().removeObject(this);

}

@Override

**public** **void** onUpdate(**float** time) {

//System.out.println(Arrays.toString(this.getWorldVerticesArr()));

}

@Override

**public** **void** onUpdateFinish(**float** time) {

*clReleaseMemObject*(**this**.worldVertices);

**this**.worldVertices = **null**;

**this**.worldVerticesArr = **null**;

}

@Override

**public** **void** onSpawn(GameObjects objectsManager) {

}

@Override

**public** **void** onDespawn(GameObjects objectsManager) {

//CL.clReleaseMemObject(this.worldVertices);

}

**public** **void** setWorldVerticesMem(cl\_mem in) {

**this**.worldVertices = in;

}

**public** cl\_mem getWorldVertices() {

**return** **this**.worldVertices;

}

/\*\*Returns the world vertices float array.<br>

\* NOTE: this is fetching the floats from the GPU

\* **@return**

\*/

**public** **float**[] getWorldVerticesArr() {

//float[] out = new float[super.mesh.getVertices().length];

//if(this.worldVerticesArr == null)

// this.worldVerticesArr = new float[super.mesh.getVertices().length];

//if(this.worldVertices == null)

// return this.worldVerticesArr;

//System.out.println(this.worldVertices);

**if**(**this**.worldVerticesArr == **null**) {

**this**.worldVerticesArr = **new** **float**[**super**.mesh.getVertices().length];

*clEnqueueReadBuffer*(

CLManager.*getCommandQueue*(),

**this**.worldVertices,

***CL\_TRUE***,

0,

Sizeof.***cl\_float*** \* **this**.worldVerticesArr.length,

Pointer.*to*(**this**.worldVerticesArr),

0,

**null**,

**null**

);

*clFinish*(CLManager.*getCommandQueue*());

}

**return** **this**.worldVerticesArr;

}

}

### Projectile.java

**package** net.alevel.asteroids.game.physics;

**import** net.alevel.asteroids.engine.graphics.Mesh;

/\*\*Represents a projectile. It uses projectile motion equations to calculate its position each update.

\*/

**public** **class** Projectile **extends** RigidObject {

**public** **static** **float** *ACC\_GRAV* = -9.8f;

**private** **float** C; //angle between where it is pointing and the x axis (between 0 and 90 degrees)

**private** **float** D; //angle between where it is pointing at and the direction towards x=+infinity. Therefore change in X is cosC and change in Z will be sinC (between 0 and 359 degrees)

**private** **float** U; //initial projection speed

//private float t;

**public** Projectile(Mesh mesh) {

**super**(mesh);

**this**.U = 0.5f;

**this**.C = 0;

**this**.D = 0;

**super**.position.set(0, 0, 0);

//this.t = 0;

}

@Override

**public** **void** onUpdate(**float** t) {

**if**(t < 0.05)

**return**;

t -= 0.05;

**super**.position.x += (**float**) ((**this**.U \* Math.*cos*(**this**.C)) \* t \* Math.*cos*(**this**.D));

**super**.position.y += (**float**) ((t \* **this**.U \* Math.*sin*(**this**.C)) + ((*ACC\_GRAV* / 2) \* Math.*pow*(t, 2)));

**super**.position.z += (**float**) ((**this**.U \* Math.*cos*(**this**.C)) \* t \* Math.*sin*(**this**.D));

//System.out.println(this + " --- " + t);

}

@Override

**public** String toString() {

**return** **super**.hashCode() + ": " + **super**.position;

}

**public** Projectile setVerticalAngleProjected(**float** C) {

**this**.C = (**float**) Math.*toRadians*(C);

**return** **this**;

}

**public** Projectile setHorizontalAngleProjected(**float** D) {

**this**.D = (**float**) Math.*toRadians*(D);

**return** **this**;

}

**public** Projectile setProjectionSpeed(**float** U) {

**this**.U = U;

**return** **this**;

}

}

### Physics.java

**package** net.alevel.asteroids.game.physics;

**import** java.io.IOException;

**import** java.util.List;

**import** net.alevel.asteroids.game.physics.SATJava.SATJava;

**import** net.alevel.asteroids.game.physics.pipeline.FunctionPipeline;

**import** net.alevel.asteroids.game.physics.pipeline.PipelineBuffer;

**import** net.alevel.asteroids.game.physics.worldCoords.WorldCoordinates;

/\*\*This manages all physics.

\*/

**public** **class** Physics {

**private** **final** PipelineBuffer physicsPipeline;

**private** **final** WorldCoordinates worldCoordsCalc;

**private** **final** FunctionPipeline collisionDetector;

**public** Physics() **throws** IOException {

**this**.physicsPipeline = **new** PipelineBuffer();

**this**.worldCoordsCalc = **new** WorldCoordinates();

//this.physicsPipeline.add(0, this.worldCoordsCalc);

**this**.collisionDetector = **new** SATJava(**this**.physicsPipeline);

}

**public** **void** onUpdate(List<RigidObject> objects) {

**if**(objects.size() == 0)

**return**;

**this**.worldCoordsCalc.calc(objects, **this**.physicsPipeline);

**this**.collisionDetector.runPipeline(objects);

}

}

## net.alevel.asteroids.game.physics.pipeline

### Releasable.java

**package** net.alevel.asteroids.game.physics.pipeline;

/\*\*Any class that implements this can be added to a {@link PipelineBuffer}. It makes sure that at the end of the pipeline, the buffer can be reset safely

\*/

**public** **interface** Releasable {

**public** **void** release();

}

### PipelineBuffer.java

**package** net.alevel.asteroids.game.physics.pipeline;

**import** java.util.HashMap;

**import** java.util.Map;

/\*\*Stores data in a map for each pipeline (or all pipelines if it is the global pipeline).

\* It is used by {@link PipelineableFunction} classes to store and communicate data between each other within a pipeline

\*/

**public** **class** PipelineBuffer {

**private** **final** Map<Integer, Releasable> releasableBuffers;

**private** **final** Map<Integer, Object> nonReleasableBuffers;

**public** PipelineBuffer() {

**this**.releasableBuffers = **new** HashMap<Integer, Releasable>();

**this**.nonReleasableBuffers = **new** HashMap<Integer, Object>();

}

**public** Object get(**int** i) {

**return** **this**.releasableBuffers.containsKey(i) ? **this**.releasableBuffers.get(i) : **this**.nonReleasableBuffers.get(i);

}

**public** **void** add(**int** i, Releasable buf) {

**if**(**this**.releasableBuffers.containsKey(i) || **this**.nonReleasableBuffers.containsKey(i))

**throw** **new** RuntimeException("The index " + i + " is already in use");

**this**.releasableBuffers.put(i, buf);

}

**public** **void** add(**int** i, Object buf) {

**if**(**this**.releasableBuffers.containsKey(i) || **this**.nonReleasableBuffers.containsKey(i))

**throw** **new** RuntimeException("The index " + i + " is already in use");

**this**.nonReleasableBuffers.put(i, buf);

}

**public** **void** release(**int** i) {

**this**.releasableBuffers.get(i).release();

**this**.releasableBuffers.remove(i);

}

**public** **void** releaseAll() {

**for**(Releasable i : **this**.releasableBuffers.values())

i.release();

**this**.releasableBuffers.clear();

}

}

### PipelineableFunction.java

**package** net.alevel.asteroids.game.physics.pipeline;

**import** java.util.List;

**import** net.alevel.asteroids.game.physics.RigidObject;

/\*\*A class that implements this can be added to a {@link FunctionPipeline}

\*/

**public** **interface** PipelineableFunction {

/\*\*The function to be run in the pipeline

\* **@param** pipelineBuffer the buffer specific to this pipeline

\* **@param** globalPipelineBuffer the buffer available to all pipelines

\* **@param** rigidObjects the in game objects

\*/

**public** **void** pipeFunction(PipelineBuffer pipelineBuffer, PipelineBuffer globalPipelineBuffer, List<RigidObject> rigidObjects);

}

### PipelineableCLMem.java

**package** net.alevel.asteroids.game.physics.pipeline;

**import** **static** org.jocl.CL.*clReleaseMemObject*;

**import** org.jocl.cl\_mem;

/\*\*A wrapper for {@link cl\_mem} that allows it to be used in a {@link PipelineBuffer}

\*/

**public** **class** PipelineableClMem **implements** Releasable {

**private** cl\_mem buffer;

**public** PipelineableClMem(cl\_mem mem) {

**this**.buffer = mem;

}

@Override

**public** **void** release() {

*clReleaseMemObject*(**this**.buffer);

}

**public** cl\_mem getBuffer() {

**return** **this**.buffer;

}

}

### PipelineableCLFunction.java

**package** net.alevel.asteroids.game.physics.pipeline;

**import** org.jocl.cl\_command\_queue;

**import** org.jocl.cl\_context;

**import** org.jocl.cl\_program;

**import** net.alevel.asteroids.game.physics.SAT.SAT;

/\*\*A {@link PipelineableFunction} but contains attributes specifically for OpenCL related functions

\*/

**public** **abstract** **class** PipelineableCLFunction **implements** PipelineableFunction {

**protected** **final** cl\_program program;

**protected** **final** cl\_context context;

**protected** **final** cl\_command\_queue commandQueue;

**public** PipelineableCLFunction(SAT pipelineController) {

**this**.program = pipelineController.getProgram();

**this**.context = pipelineController.getContext();

**this**.commandQueue = pipelineController.getCommandQueue();

}

}

### FunctionPipeline.java

**package** net.alevel.asteroids.game.physics.pipeline;

**import** java.util.Iterator;

**import** java.util.List;

**import** net.alevel.asteroids.game.physics.RigidObject;

/\*\*Represents a pipeline. It is a list of functions that are run sequentially and use a {@link PipelineBuffer} or several to store and exchange data between each pipeline stage.

\*/

**public** **class** FunctionPipeline **implements** Iterable<PipelineableFunction> {

**private** PipelineableFunction[] functionPipeline;

**private** **final** PipelineBuffer bufferPipeline;

**private** **final** PipelineBuffer globalBufferPipeline;

**public** FunctionPipeline(PipelineBuffer globalPipeline, PipelineableFunction... functions) {

**this**.functionPipeline = functions;

**this**.globalBufferPipeline = globalPipeline;

**this**.bufferPipeline = **new** PipelineBuffer();

}

**protected** **void** setFunctions(PipelineableFunction... functions) {

**this**.functionPipeline = functions;

}

**public** **void** runPipeline(List<RigidObject> rigidObjects) {

**this**.bufferPipeline.releaseAll();

**for**(PipelineableFunction i : **this**.functionPipeline)

i.pipeFunction(**this**.bufferPipeline, **this**.globalBufferPipeline, rigidObjects);

}

@Override

**public** Iterator<PipelineableFunction> iterator() {

**return** **new** Iterator<PipelineableFunction>() {

**private** **int** pointer = 0;

@Override

**public** **boolean** hasNext() {

**return** pointer < functionPipeline.length;

}

@Override

**public** PipelineableFunction next() {

**return** functionPipeline[pointer++];

}

@Override

**public** **void** remove() {

**throw** **new** UnsupportedOperationException("Cannot remove functions from pipeline!");

}

};

}

**public** PipelineBuffer getBufferPipeline() {

**return** **this**.bufferPipeline;

}

}

## net.alevel.asteroids.game.physics.SAT

### ProjectedBoundaries.java

**package** net.alevel.asteroids.game.physics.SAT;

**import** **static** org.jocl.CL.\*;

**import** java.util.List;

**import** org.jocl.Pointer;

**import** org.jocl.Sizeof;

**import** org.jocl.cl\_kernel;

**import** org.jocl.cl\_mem;

**import** net.alevel.asteroids.game.physics.RigidObject;

**import** net.alevel.asteroids.game.physics.SATJava.SATJava;

**import** net.alevel.asteroids.game.physics.pipeline.PipelineableClMem;

**import** net.alevel.asteroids.game.physics.pipeline.PipelineBuffer;

**import** net.alevel.asteroids.game.physics.pipeline.PipelineableCLFunction;

**import** net.alevel.asteroids.game.physics.worldCoords.WorldCoordinates;

/\*\*Calculates the projected boundaries for each axis. This uses the same mathematical equation that I created in {@link SATJava}.

\*/

**public** **class** ProjectedBoundaries **extends** PipelineableCLFunction {

**private** **final** cl\_kernel projectPointsKernel;

**private** **final** cl\_kernel boundariesKernel;

**public** ProjectedBoundaries(SAT sat) {

**super**(sat);

**this**.projectPointsKernel = *clCreateKernel*(**super**.program, "getProjectedVertices", **null**);

**this**.boundariesKernel = *clCreateKernel*(**super**.program, "getBoundaries", **null**);

}

@Override

**public** **void** pipeFunction(PipelineBuffer pipelineBuffer, PipelineBuffer globalPipelineBuffer, List<RigidObject> rigidObjects) {

*setExceptionsEnabled*(**true**);

cl\_mem surfaceNormalsBuffer = ((PipelineableClMem) pipelineBuffer.get(0)).getBuffer();

**int** noOfSurfaceNormals = (**int**) pipelineBuffer.get(1);

WorldCoordinates worldCoords = (WorldCoordinates) globalPipelineBuffer.get(0);

**int**[] subBufferPointers = worldCoords.getSubBufferPointers();

//cl\_mem boundaries = clCreateBuffer(super.context, CL\_MEM\_READ\_WRITE, rigidObjects.size() \* noOfSurfaceNormals \* 2 \* Sizeof.cl\_float, null, null);

cl\_mem maxBoundariesTemp = *clCreateBuffer*(**super**.context, ***CL\_MEM\_READ\_WRITE***, worldCoords.getNoOfVertices() \* noOfSurfaceNormals \* Sizeof.***cl\_float***, **null**, **null**);

cl\_mem minBoundariesTemp = *clCreateBuffer*(**super**.context, ***CL\_MEM\_READ\_WRITE***, worldCoords.getNoOfVertices() \* noOfSurfaceNormals \* Sizeof.***cl\_float***, **null**, **null**);

//cl\_mem subBufPointBuffer = clCreateBuffer(super.context, CL\_MEM\_READ\_ONLY | CL\_MEM\_COPY\_HOST\_PTR, subBufferPointers.length \* Sizeof.cl\_int, Pointer.to(subBufferPointers), null);

*clSetKernelArg*(**this**.projectPointsKernel, 0, Sizeof.***cl\_mem***, Pointer.*to*(surfaceNormalsBuffer));

*clSetKernelArg*(**this**.projectPointsKernel, 1, Sizeof.***cl\_mem***, Pointer.*to*(worldCoords.getWorldCoords()));

*clSetKernelArg*(**this**.projectPointsKernel, 2, Sizeof.***cl\_mem***, Pointer.*to*(minBoundariesTemp));

*clSetKernelArg*(**this**.projectPointsKernel, 3, Sizeof.***cl\_mem***, Pointer.*to*(maxBoundariesTemp));

*clSetKernelArg*(**this**.boundariesKernel, 0, Sizeof.***cl\_mem***, Pointer.*to*(maxBoundariesTemp));

*clSetKernelArg*(**this**.boundariesKernel, 1, Sizeof.***cl\_mem***, Pointer.*to*(minBoundariesTemp));

//clSetKernelArg(this.boundariesKernel, 2, Sizeof.cl\_mem, Pointer.to(subBufPointBuffer));

//clSetKernelArg(this.boundariesKernel, 3, Sizeof.cl\_mem, Pointer.to(boundaries));

*clEnqueueNDRangeKernel*(

**super**.commandQueue,

**this**.projectPointsKernel,

1,

**null**,

**new** **long**[] {noOfSurfaceNormals},

**new** **long**[] {worldCoords.getNoOfVertices() \* noOfSurfaceNormals},

0,

**null**,

**null**

);

*clFinish*(**super**.commandQueue);

**for**(**int** i = 0; i < subBufferPointers.length; i++) {

*clEnqueueNDRangeKernel*(

**super**.commandQueue,

**this**.boundariesKernel,

1,

**new** **long**[] {subBufferPointers[i]},

**new** **long**[] {(i + 1) == subBufferPointers.length ? worldCoords.getNoOfVertices() - subBufferPointers[i] : subBufferPointers[i + 1] - subBufferPointers[i]},

**new** **long**[] {worldCoords.getNoOfVertices()},

0,

**null**,

**null**);

}

//clFinish(super.commandQueue);

**float**[] maxBoundariesArray = **new** **float**[worldCoords.getNoOfVertices() \* noOfSurfaceNormals];

**float**[] minBoundariesArray = **new** **float**[worldCoords.getNoOfVertices() \* noOfSurfaceNormals];

*clEnqueueReadBuffer*(

**super**.commandQueue,

maxBoundariesTemp,

***CL\_TRUE***,

0,

Sizeof.***cl\_float*** \* maxBoundariesArray.length,

Pointer.*to*(maxBoundariesArray),

0,

**null**,

**null**);

*clEnqueueReadBuffer*(

**super**.commandQueue,

minBoundariesTemp,

***CL\_TRUE***,

0,

Sizeof.***cl\_float*** \* minBoundariesArray.length,

Pointer.*to*(minBoundariesArray),

0,

**null**,

**null**);

*clReleaseMemObject*(minBoundariesTemp);

*clReleaseMemObject*(maxBoundariesTemp);

}

}

### SurfaceNormals.java

**package** net.alevel.asteroids.game.physics.SAT;

**import** **static** org.jocl.CL.***CL\_MEM\_COPY\_HOST\_PTR***;

**import** **static** org.jocl.CL.***CL\_MEM\_READ\_ONLY***;

**import** **static** org.jocl.CL.***CL\_MEM\_READ\_WRITE***;

**import** **static** org.jocl.CL.***CL\_TRUE***;

**import** **static** org.jocl.CL.*clCreateBuffer*;

**import** **static** org.jocl.CL.*clCreateKernel*;

**import** **static** org.jocl.CL.*clEnqueueNDRangeKernel*;

**import** **static** org.jocl.CL.*clEnqueueReadBuffer*;

**import** **static** org.jocl.CL.*clSetKernelArg*;

**import** **static** org.jocl.CL.*setExceptionsEnabled*;

**import** java.io.IOException;

**import** java.util.Arrays;

**import** java.util.List;

**import** org.jocl.Pointer;

**import** org.jocl.Sizeof;

**import** org.jocl.cl\_kernel;

**import** org.jocl.cl\_mem;

**import** net.alevel.asteroids.game.physics.RigidObject;

**import** net.alevel.asteroids.game.physics.pipeline.PipelineableClMem;

**import** net.alevel.asteroids.game.physics.pipeline.PipelineBuffer;

**import** net.alevel.asteroids.game.physics.pipeline.PipelineableCLFunction;

**import** net.alevel.asteroids.game.physics.worldCoords.WorldCoordinates;

/\*\*Calculates all the surface normals in the world. These will be the axis.

\* It takes the world coordinates from the pipeline buffer (which is 1 float buffer containing all the world coordinates for all objects)

\* and calculates the surface normals using sub buffer pointers which point to the first index of each object's set of world coordinates in the buffer

\*/

**public** **class** SurfaceNormals **extends** PipelineableCLFunction {

**private** **final** cl\_kernel surfaceNormalKernel;

**public** SurfaceNormals(SAT sat) **throws** IOException {

**super**(sat);

**this**.surfaceNormalKernel = *clCreateKernel*(**super**.program, "getSurfaceNormals", **null**);

}

@Override

**public** **void** pipeFunction(PipelineBuffer pipelineBuffer, PipelineBuffer globalPipelineBuffer, List<RigidObject> rigidObjects) {

*setExceptionsEnabled*(**true**);

WorldCoordinates worldCoordinates = (WorldCoordinates) globalPipelineBuffer.get(0);

**int** noOfIndices = 0;

**for**(RigidObject i : rigidObjects)

noOfIndices += i.getMesh().getIndices().length;

**int**[] indicesArray = **new** **int**[noOfIndices];

**int**[] worldCoordSubBufferPointers = worldCoordinates.getSubBufferPointers();

**for**(**int** i = 0; i < rigidObjects.size(); i++) {

**int**[] rigidObjectIndicesTemp = rigidObjects.get(i).getMesh().getIndices();

**for**(**int** j = 0; j < rigidObjectIndicesTemp.length; j++)

indicesArray[worldCoordSubBufferPointers[i] + j] = rigidObjectIndicesTemp[j] + worldCoordSubBufferPointers[i];

}

cl\_mem indicesBuffer = *clCreateBuffer*(**super**.context, ***CL\_MEM\_READ\_ONLY*** | ***CL\_MEM\_COPY\_HOST\_PTR***, Sizeof.***cl\_int*** \* noOfIndices, Pointer.*to*(indicesArray), **null**);

cl\_mem surfaceNormalsBuffer = *clCreateBuffer*(**super**.context, ***CL\_MEM\_READ\_WRITE***, Sizeof.***cl\_float*** \* noOfIndices, **null**, **null**);

*clSetKernelArg*(**this**.surfaceNormalKernel, 0, Sizeof.***cl\_mem***, Pointer.*to*(worldCoordinates.getWorldCoords()));

*clSetKernelArg*(**this**.surfaceNormalKernel, 1, Sizeof.***cl\_mem***, Pointer.*to*(indicesBuffer));

*clSetKernelArg*(**this**.surfaceNormalKernel, 2, Sizeof.***cl\_mem***, Pointer.*to*(surfaceNormalsBuffer));

**long**[] global\_work\_size = {noOfIndices / 3};

*clEnqueueNDRangeKernel*(

**super**.commandQueue,

**this**.surfaceNormalKernel,

global\_work\_size.length,

**null**,

global\_work\_size,

**null**,

0,

**null**,

**null**);

**float**[] surfaceNormalsArray = **new** **float**[noOfIndices];

*clEnqueueReadBuffer*(

**super**.commandQueue,

surfaceNormalsBuffer,

***CL\_TRUE***,

0,

Sizeof.***cl\_float*** \* surfaceNormalsArray.length,

Pointer.*to*(surfaceNormalsArray),

0,

**null**,

**null**);

System.***out***.println(Arrays.*toString*(surfaceNormalsArray));

pipelineBuffer.add(0, **new** PipelineableClMem(surfaceNormalsBuffer));

pipelineBuffer.add(1, noOfIndices / 3); //number of surface normals needed for other pipeline functions

}

}

### SAT.java

**package** net.alevel.asteroids.game.physics.SAT;

**import** **static** net.alevel.asteroids.game.cl.CLUtil.*loadProgram*;

**import** java.io.IOException;

**import** org.jocl.cl\_command\_queue;

**import** org.jocl.cl\_context;

**import** org.jocl.cl\_program;

**import** net.alevel.asteroids.game.cl.CLManager;

**import** net.alevel.asteroids.game.physics.pipeline.FunctionPipeline;

**import** net.alevel.asteroids.game.physics.pipeline.PipelineBuffer;

/\*\*This class contains the code for the OpenCL implementation of the separating axis theorem.

\* Note on local pipeline buffer<br>

\* 0 -> surface normals<br>

\* 1 -> projectedBoundaries<br>

\* <br>

\* World coordinates are in global pipeline (0) because they may be used in other pipelines

\*/

**public** **class** SAT **extends** FunctionPipeline {

**private** **final** cl\_context context;

**private** **final** cl\_command\_queue commandQueue;

**private** **final** cl\_program program;

**public** SAT(PipelineBuffer globalBufferPipeline) **throws** IOException {

**super**(globalBufferPipeline);

**this**.context = CLManager.*getContext*();

**this**.commandQueue = CLManager.*getCommandQueue*();

**this**.program = *loadProgram*("SAT.cl", SAT.**class**, **this**.context);

**super**.setFunctions(

**new** SurfaceNormals(**this**),

**new** ProjectedBoundaries(**this**)

);

}

/\*@Override

public void runPipeline(List<RigidObject> rigidObjects) {

//if(rigidObjects.size() < 2)

// return;

super.runPipeline(rigidObjects);

}\*/

**public** cl\_context getContext() {

**return** **this**.context;

}

**public** cl\_command\_queue getCommandQueue() {

**return** **this**.commandQueue;

}

**public** cl\_program getProgram() {

**return** **this**.program;

}

}

### SAT.cl

//this file contains functions for my OpenCL SAT (Separating axis theorem) implementation

//All world coordinates are stored in one huge array so the program could scale well and use the GPU to its full SIMT (same instruction multiple threads) capacity.

//int get\_group\_size(int x);

//This kernel is used in SAT.SurfaceNormals. It calculates a surface normal for each triangle for all triangles in the world. Each kernel instance is responsible for 1 triangle

kernel void getSurfaceNormals(

    global const float \*vertices, //vector3f array (These are the transformed vertices (world coords, not model))

                                    //i.e. the rotation and position vectors have been applied.

    global const int \*indices, //vector3i array where vertices x, y and z are corners of a triangle

    global float \*surfaceNormals //vector3f array

)

{

    int globalId = get\_global\_id(0) \* 3;

    float3 a = (float3)(

        vertices[indices[globalId + 1] \* 3] - vertices[indices[globalId] \* 3],

        vertices[indices[globalId + 1] \* 3 + 1] - vertices[indices[globalId] \* 3 + 1],

        vertices[indices[globalId + 1] \* 3 + 2] - vertices[indices[globalId] \* 3 + 2]

    );

    float3 b = (float3)(

        vertices[indices[globalId + 2] \* 3] - vertices[indices[globalId] \* 3],

        vertices[indices[globalId + 2] \* 3 + 1] - vertices[indices[globalId] \* 3 + 1],

        vertices[indices[globalId + 2] \* 3 + 2] - vertices[indices[globalId] \* 3 + 2]

    );

    float3 c = normalize(cross(a, b));

    surfaceNormals[globalId] = c.x;

    surfaceNormals[globalId + 1] = c.y;

    surfaceNormals[globalId + 2] = c.z;

}

//These project the vertices onto each axis (surface normal)

kernel void getProjectedVertices(

    global const float \*surfaceNormals,

    global const float \*vertices,

    global float \*projectedVerticesMin, //these arrays store exactly the same data which is needed for how I have done the getBoundaries method

    global float \*projectedVerticesMax

)

{

    int localId = get\_local\_id(0) \* 3;

    int groupId = get\_group\_id(0) \* 3;

    int groupSize = get\_group\_size(0);

    float3 a = (float3)(vertices[localId], vertices[localId + 1], vertices[localId + 2]);

    float3 n = (float3)(surfaceNormals[groupId], surfaceNormals[groupId + 1], surfaceNormals[groupId + 2]);

    float lambda = dot(a, n) / (pow(n.x, 2) + pow(n.y, 2) + pow(n.z, 2));

    int projectedVerticesIndex = get\_group\_id(0) \* groupSize + get\_local\_id(0);

    projectedVerticesMin[projectedVerticesIndex] = lambda;

    projectedVerticesMax[projectedVerticesIndex] = lambda;

    /\*projectedVerticesMin[projectedVerticesIndex] = lambda \* n.x;

    projectedVerticesMin[projectedVerticesIndex + 1] = lambda \* n.y;

    projectedVerticesMin[projectedVerticesIndex + 2] = lambda \* n.z;

    projectedVerticesMax[projectedVerticesIndex] = lambda \* n.x;

    projectedVerticesMax[projectedVerticesIndex + 1] = lambda \* n.y;

    projectedVerticesMax[projectedVerticesIndex + 2] = lambda \* n.z;\*/

}

//Gets the max and min values for lambda for each axis for each object. To be used when testing collisions as 2 intersecting. It uses a parallel reduction algorithm (https://dournac.org/info/gpu\_sum\_reduction)

kernel void getBoundaries(

    global float \*maxProjectedBoundaries,

    global float \*minProjectedBoundaries

)

{

    int globalId = get\_global\_id(0);

    int groupId = get\_group\_id(0);

    int localId = get\_local\_id(0);

    int groupSize = get\_group\_size(0);

    //get minimum

    barrier(CLK\_LOCAL\_MEM\_FENCE);

    //int trueGlobalId = localId; // + get\_global\_offset(0); //globalId - get\_global\_offset(0);

    int index = groupId \* groupSize + localId;

    for(int stride = groupSize >> 1; stride != 0; stride >>= 1) { //int stride = groupSize / 2; stride > 0; stride /= 2

        if(localId < stride) {

            int a = minProjectedBoundaries[index];

            int b = minProjectedBoundaries[index + stride];

            if(b < a){

                minProjectedBoundaries[index] = b;

            }

        }

        barrier(CLK\_LOCAL\_MEM\_FENCE);

    }

    if(localId == 0){

        /\*int offset = get\_global\_offset(0);

        int subBufferIndex = 0;

        for(; subBufferPointers[subBufferIndex] != offset; subBufferIndex++);

        int boundaryIndex = (groupId \* groupSize + subBufferIndex) \* 2;

        boundaries[boundaryIndex] = projectedVerticesMin[0];\*/

    }

    //same algorithm but for max value

    //barrier(CLK\_LOCAL\_MEM\_FENCE);

    for(int stride = groupSize >> 1; stride != 0; stride >>= 1) {

        if(localId < stride) {

            int a = maxProjectedBoundaries[globalId];

            int b = maxProjectedBoundaries[globalId + stride];

            if(b > a){

                maxProjectedBoundaries[globalId] = b;

            }

        }

        barrier(CLK\_LOCAL\_MEM\_FENCE);

    }

    if(localId == 0){

        //int boundaryIndex = (localId + groupId) \* 2;

        //boundaries[boundaryIndex + 1] = projectedVerticesMax[0];

    }

}

//use the boundaries to check for collisions from different objects

/\*kernel void testIntersections(

    global const float \*boundaries,

    global float \*collisions

)

{

}\*/

## net.alevel.asteroids.game.physics.SATJava

### SATJava.java

**package** net.alevel.asteroids.game.physics.SATJava;

**import** **static** java.lang.Math.*pow*;

**import** java.util.Arrays;

**import** java.util.HashSet;

**import** java.util.List;

**import** java.util.Set;

**import** org.joml.Vector3f;

**import** net.alevel.asteroids.game.physics.RigidObject;

**import** net.alevel.asteroids.game.physics.pipeline.FunctionPipeline;

**import** net.alevel.asteroids.game.physics.pipeline.PipelineBuffer;

**import** net.alevel.asteroids.game.physics.pipeline.PipelineableFunction;

/\*\*A simple implementation of the 'Separating Axis Theorem (SAT)' algorithm. This is a common method of collision detection in games.

\* Firstly, it calculates a normal vector for each triangle. These will be used as the axis that each object.<br>

\* <br>The projection is done like this:

\* <br>let 'A' be The world coordinate of the vertex to project

\* <br>let 'n' the axis to project onto. The axis is treated as a 3D line going through the origin. n is the direction of this line

\* <br>let 'B' the point on the line where the line AB is perpendicular to the line 'n'

\* <br>then B = lambda \* n

\* <br>AB = B - A

\* <br>dot(AB, n) = 0

\* <br>n.x \* (n.x \* lambda - A.x) + n.y \* (n.y \* lambda - A.y) + n.z \* (n.z \* lambda - A.z) = 0

\* <br>pow(n.x, 2) \* lambda - A.x \* n.x + pow(n.y, 2) \* lambda - A.y \* n.y + pow(n.z, 2) \* lambda - A.z \* n.z = 0

\* <br>lambda \* (pow(n.x, 2) + pow(n.y, 2) + pow(n.z, 2)) = A.x \* n.x + A.y \* n.y + A.z \* n.z

\* <br>lambda = dot(A, n) / pow(length(n), 2)

\*/

**public** **class** SATJava **extends** FunctionPipeline **implements** PipelineableFunction {

**private** **boolean** calc;

**public** SATJava(PipelineBuffer globalPipeline) {

**super**(globalPipeline);

**super**.setFunctions(**this**);

**this**.calc = **false**;

}

@Override

**public** **void** pipeFunction(PipelineBuffer pipelineBuffer, PipelineBuffer globalPipelineBuffer, List<RigidObject> rigidObjects) {

//calculate surface normals (these will be the axis used)

**if**(!**this**.calc) {

**this**.calc = **true**;

**return**;

}

System.***out***.println(rigidObjects);

Set<Vector3f> surfaceNormals = **new** HashSet<Vector3f>();

**for**(RigidObject rigidObject : rigidObjects) {

**float**[] vertices = rigidObject.getWorldVerticesArr();

System.***out***.println(Arrays.*toString*(vertices));

**int**[] indices = rigidObject.getMesh().getIndices();

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

Vector3f a = **new** Vector3f(

vertices[indices[i + 1] \* 3] - vertices[indices[i] \* 3],

vertices[indices[i + 1] \* 3 + 1] - vertices[indices[i] \* 3 + 1],

vertices[indices[i + 1] \* 3 + 2] - vertices[indices[i] \* 3 + 2]

);

Vector3f b = **new** Vector3f(

vertices[indices[i + 2] \* 3] - vertices[indices[i] \* 3],

vertices[indices[i + 2] \* 3 + 1] - vertices[indices[i] \* 3 + 1],

vertices[indices[i + 2] \* 3 + 2] - vertices[indices[i] \* 3 + 2]

);

Vector3f c = a.cross(b).normalize();

//Vector3f c2 = new Vector3f(c).mul(-1);

//System.out.println(c + " " + c2 + " " + surfaceNormals.contains(c) + " " + surfaceNormals.contains(c2));

**if**(!**this**.alreadyHasOppositeDirection(c, surfaceNormals))

//if(!surfaceNormals.contains(new Vector3f(c).mul(-1)))

surfaceNormals.add(c);

}

}

//System.out.println();

//System.out.println(surfaceNormals);

//calculate the projected boundaries for each normal (axis)

ObjectProjection[] projectedVertices = **new** ObjectProjection[rigidObjects.size()];

**for**(**int** a = 0; a < rigidObjects.size(); a++) {

**float**[] allRanges = **new** **float**[surfaceNormals.size() \* 2];

**int** counter = 0;

**for**(Vector3f normal : surfaceNormals) {

**float**[] vertices = rigidObjects.get(a).getWorldVerticesArr();

**float** modNSquared = (**float**) (*pow*(normal.x, 2) + *pow*(normal.y, 2) + *pow*(normal.z, 2));

**float** lambda = ((vertices[0] \* normal.x) + (vertices[1] \* normal.y) + (vertices[2] \* normal.z)) / modNSquared;

**float** min = lambda,

max = lambda;

**for**(**int** i = 3; i < vertices.length; i += 3) {

lambda = ((vertices[i] \* normal.x) + (vertices[i + 1] \* normal.y) + (vertices[i + 2] \* normal.z)) / modNSquared;

**if**(lambda < min)

min = lambda;

**else** **if**(lambda > max)

max = lambda;

}

allRanges[counter] = min;

allRanges[counter + 1] = max;

counter += 2;

}

projectedVertices[a] = **new** ObjectProjection(rigidObjects.get(a), allRanges);

System.***out***.println(surfaceNormals);

System.***out***.println(Arrays.*toString*(allRanges) + "\n");

}

//test collisions

**for**(**int** i = 0; i < projectedVertices.length; i++)

**for**(**int** j = i + 1; j < projectedVertices.length; j++)

testCollision(projectedVertices[i], projectedVertices[j]);

}

**private** **boolean** alreadyHasOppositeDirection(Vector3f v, Set<Vector3f> set) {

**for**(Vector3f i : set) {

//System.out.println(v + " " + i + " " + (i.x + v.x == 0f && i.y + v.y == 0f && i.z + v.z == 0f));

**if**(i.x + v.x == 0f && i.y + v.y == 0f && i.z + v.z == 0f) {

//System.out.println(0);

**return** **true**;

}

}

**return** **false**;

}

**private** **void** testCollision(ObjectProjection o1, ObjectProjection o2) {

**for**(**int** i = 0; i < o1.projections.length; i += 2) {

System.***out***.println(o1.projections[i + 1] < o2.projections[i] || o2.projections[i + 1] < o1.projections[i]);

**if**(o1.projections[i + 1] < o2.projections[i] || o2.projections[i + 1] < o1.projections[i])

**return**;

}

System.***out***.println("Collision!");

o1.object.onCollision(o2.object);

o2.object.onCollision(o1.object);

}

**private** **class** ObjectProjection {

**private** RigidObject object;

**private** **float**[] projections;

**public** ObjectProjection(RigidObject o, **float**[] p) {

**this**.object = o;

**this**.projections = p;

}

}

}

## net.alevel.asteroids.game.physics.worldCoords

### WorldCoordinates.java

**package** net.alevel.asteroids.game.physics.worldCoords;

**import** **static** net.alevel.asteroids.game.cl.CLUtil.*loadProgram*;

**import** **static** org.jocl.CL.***CL\_BUFFER\_CREATE\_TYPE\_REGION***;

**import** **static** org.jocl.CL.***CL\_MEM\_COPY\_HOST\_PTR***;

**import** **static** org.jocl.CL.***CL\_MEM\_READ\_ONLY***;

**import** **static** org.jocl.CL.***CL\_MEM\_READ\_WRITE***;

**import** **static** org.jocl.CL.*clCreateBuffer*;

**import** **static** org.jocl.CL.*clCreateKernel*;

**import** **static** org.jocl.CL.*clCreateSubBuffer*;

**import** **static** org.jocl.CL.*clEnqueueNDRangeKernel*;

**import** **static** org.jocl.CL.*clFinish*;

**import** **static** org.jocl.CL.*clReleaseMemObject*;

**import** **static** org.jocl.CL.*clSetKernelArg*;

**import** **static** org.jocl.CL.*setExceptionsEnabled*;

**import** java.io.IOException;

**import** java.util.List;

**import** org.jocl.Pointer;

**import** org.jocl.Sizeof;

**import** org.jocl.cl\_buffer\_region;

**import** org.jocl.cl\_command\_queue;

**import** org.jocl.cl\_context;

**import** org.jocl.cl\_kernel;

**import** org.jocl.cl\_mem;

**import** org.jocl.cl\_program;

**import** org.joml.Vector3f;

**import** net.alevel.asteroids.game.cl.CLManager;

**import** net.alevel.asteroids.game.physics.RigidObject;

**import** net.alevel.asteroids.game.physics.pipeline.PipelineBuffer;

**import** net.alevel.asteroids.game.physics.pipeline.Releasable;

/\*\*Calculates the world coordinates of each model vertex.

\* In other words, it applies an object's position, scale, and rotation to work out where the positions are of each vertex in its mesh.<br>

\* This uses OpenCL. It calculates the world coordinates on a GPU (or whatever device is specified by {@link CLManager}) as each vertex position can be calculated in parallel.

\*/

**public** **class** WorldCoordinates **implements** Releasable {

**private** **final** cl\_program program;

**private** **final** cl\_kernel rotMatKernel;

**private** **final** cl\_kernel transformKernel;

**private** **final** cl\_context context;

**private** **final** cl\_command\_queue commandQueue;

**private** cl\_mem worldCoords;

**private** cl\_mem worldCoordsIndices;

**private** **int**[] objectSubBufferPointers;

**private** **int** numberOfVertices;

**public** WorldCoordinates() **throws** IOException {

**this**.context = CLManager.*getContext*();

**this**.commandQueue = CLManager.*getCommandQueue*();

**this**.program = *loadProgram*("WorldCoordinates.cl", WorldCoordinates.**class**, **this**.context);

**this**.transformKernel = *clCreateKernel*(**this**.program, "tranformVectors", **null**);

**this**.rotMatKernel = *clCreateKernel*(**this**.program, "getRotationMatrix", **null**);

}

**public** **void** calc(List<RigidObject> objects, PipelineBuffer pipeline) {

*setExceptionsEnabled*(**true**);

**this**.numberOfVertices = 0;

**for**(RigidObject i : objects)

**this**.numberOfVertices += i.getMesh().getVertices().length;

**float**[] floats = **new** **float**[**this**.numberOfVertices];

**int**[] objectIndices = **new** **int**[**this**.numberOfVertices / 3]; //an array telling which vertex belongs to which object

**float**[] objectPositions = **new** **float**[objects.size() \* 3];

**float**[] objectRotations = **new** **float**[objects.size() \* 3]; //each object has 2 float3 vector properties (2 \* 3 = 6)

/\*E.g. floats = {1, 0, 0, 1, 1, 1}

\* objectIndices = {0, 1}

\* objectProperties = {0, 1, 0, 1, 1, 1, 0, 0, 0, 1, 1, 0}

\* It will apply the object properties of object 0 to the first vector. The second vector will have object 1's properties applied to it

\*/

**this**.numberOfVertices /= 3; //because before this number represented the size of the float array which is 3x the number of vertices

**int** i = 0, i2 = 0;

**for**(**int** objectIndex = 0; objectIndex < objects.size(); objectIndex++) {

RigidObject o = objects.get(objectIndex);

**float**[] vertices = o.getMesh().getVertices();

**for**(**int** j = 0; j < vertices.length; j += 3, i += 3, i2++) {

floats[i] = vertices[j];

floats[i + 1] = vertices[j + 1];

floats[i + 2] = vertices[j + 2];

objectIndices[i2] = objectIndex;

}

Vector3f position = o.getPosition();

Vector3f rotation = o.getRotation();

objectPositions[objectIndex \* 3] = position.x;

objectPositions[objectIndex \* 3 + 1] = position.y;

objectPositions[objectIndex \* 3 + 2] = position.z;

objectRotations[objectIndex \* 3] = rotation.x;

objectRotations[objectIndex \* 3 + 1] = rotation.y;

objectRotations[objectIndex \* 3 + 2] = rotation.z;

}

//System.out.println(floats.length);

cl\_mem floatsMem = *clCreateBuffer*(**this**.context, ***CL\_MEM\_READ\_ONLY*** | ***CL\_MEM\_COPY\_HOST\_PTR***, Sizeof.***cl\_float*** \* floats.length, Pointer.*to*(floats), **null**);

cl\_mem objectIndicesMem = *clCreateBuffer*(**this**.context, ***CL\_MEM\_READ\_ONLY*** | ***CL\_MEM\_COPY\_HOST\_PTR***, Sizeof.***cl\_int*** \* objectIndices.length, Pointer.*to*(objectIndices), **null**);

cl\_mem objectPositionsMem = *clCreateBuffer*(**this**.context, ***CL\_MEM\_READ\_ONLY*** | ***CL\_MEM\_COPY\_HOST\_PTR***, Sizeof.***cl\_float*** \* objectPositions.length, Pointer.*to*(objectPositions), **null**);

cl\_mem objectRotationsMem = *clCreateBuffer*(**this**.context, ***CL\_MEM\_READ\_ONLY*** | ***CL\_MEM\_COPY\_HOST\_PTR***, Sizeof.***cl\_float*** \* objectRotations.length, Pointer.*to*(objectRotations), **null**);

cl\_mem objectRotationsMatMem = *clCreateBuffer*(**this**.context, ***CL\_MEM\_READ\_WRITE***, Sizeof.***cl\_float*** \* objectRotations.length \* 3, **null**, **null**);

cl\_mem transformedVerticesMem = *clCreateBuffer*(**this**.context, ***CL\_MEM\_READ\_WRITE***, Sizeof.***cl\_float*** \* floats.length, **null**, **null**);

*clSetKernelArg*(**this**.rotMatKernel, 0, Sizeof.***cl\_mem***, Pointer.*to*(objectRotationsMem));

*clSetKernelArg*(**this**.rotMatKernel, 1, Sizeof.***cl\_mem***, Pointer.*to*(objectRotationsMatMem));

*clSetKernelArg*(**this**.transformKernel, 0, Sizeof.***cl\_mem***, Pointer.*to*(floatsMem));

*clSetKernelArg*(**this**.transformKernel, 1, Sizeof.***cl\_mem***, Pointer.*to*(objectIndicesMem));

*clSetKernelArg*(**this**.transformKernel, 2, Sizeof.***cl\_mem***, Pointer.*to*(objectPositionsMem));

*clSetKernelArg*(**this**.transformKernel, 3, Sizeof.***cl\_mem***, Pointer.*to*(objectRotationsMatMem));

*clSetKernelArg*(**this**.transformKernel, 4, Sizeof.***cl\_mem***, Pointer.*to*(transformedVerticesMem));

**long**[] global\_work\_size = {objectRotations.length / 3};

//long[] local\_work\_size = {3};

*clEnqueueNDRangeKernel*(

**this**.commandQueue,

rotMatKernel,

global\_work\_size.length,

**null**,

global\_work\_size,

**null**,

0,

**null**,

**null**);

//clFinish(CLManager.getCommandQueue());

global\_work\_size[0] = objectIndices.length;

*clEnqueueNDRangeKernel*(

**this**.commandQueue,

transformKernel,

global\_work\_size.length,

**null**,

global\_work\_size,

**null**,

0,

**null**,

**null**);

*clFinish*(**this**.commandQueue);

**this**.objectSubBufferPointers = **new** **int**[objects.size()];

**for**(**int** j = 0, c = 0; j < objects.size(); j++) {

RigidObject o = objects.get(j);

**int** length = o.getMesh().getVertices().length;

o.setWorldVerticesMem(*clCreateSubBuffer*(transformedVerticesMem, ***CL\_MEM\_READ\_ONLY***, ***CL\_BUFFER\_CREATE\_TYPE\_REGION***, **new** cl\_buffer\_region(Sizeof.***cl\_float*** \* c, Sizeof.***cl\_float*** \* length), **null**));

//System.out.print(c);

**this**.objectSubBufferPointers[j] = c;

c += length;

//System.out.println(" " + c);

}

*clFinish*(**this**.commandQueue); //makes sure that all vertices for all objects have been processed

//The CPU will continue the pipeline and queue more instructions regardless of whether the GPU is finished or not

//clReleaseMemObject(transformedVerticesMem);

**this**.worldCoords = transformedVerticesMem;

**this**.worldCoordsIndices = objectIndicesMem;

*clReleaseMemObject*(floatsMem);

//clReleaseMemObject(objectIndicesMem);

*clReleaseMemObject*(objectPositionsMem);

*clReleaseMemObject*(objectRotationsMem);

*clReleaseMemObject*(objectRotationsMatMem);

}

**public** **void** release() {

*clReleaseMemObject*(**this**.worldCoords);

*clReleaseMemObject*(**this**.worldCoordsIndices);

}

**public** **int**[] getSubBufferPointers() {

**return** **this**.objectSubBufferPointers;

}

**public** cl\_mem getWorldCoords() {

**return** worldCoords;

}

**public** cl\_mem getWorldCoordsIndices() {

**return** worldCoordsIndices;

}

**public** **int** getNoOfVertices() {

**return** **this**.numberOfVertices;

}

}

### WorldCoordinates.cl

/\*

This file uses a modified version of the C language optimised for parallel.

The functions here are known as kernels. They are designed in a way that allows multiple of them to run physically in parallel.

Each instance of a kernel has several unique properties (e.g. global id, local id). I use these IDs to select the input data from the

input arrays. For example, lets say inputData = {1, 2, 3, 4, 5, 6} and outputData = {0, 0, 0, 0, 0, 0}. You want to add 10 to each value in the inputData array.

The kernel would look like this:

kernel void add10(

    global int inputData\*

    global int outputData\*

)

{

    int id = get\_global\_id(0)

    outputData[id] = inputData[id] + 10;

}

There is no for loop required here. In the java code, I would just need to enqueue this kernel and tell it to do 10 instances for it. Each kernel instance

will have a unique global ID (between 0 and 9) which I can use to specify the input data and the location where to store the data.

\*/

\_\_kernel void tranformVectors(

    \_\_global const float \*modelVertices,

    \_\_global const int \*objectIndices,

    \_\_global const float \*posVectors,

    \_\_global const float \*rotMatrices, //column major format

    \_\_global float \*worldVertices

)

{

    int id = get\_global\_id(0);

    int objectIndex = objectIndices[id];

    int rotateIndex = objectIndices[id] \* 9;

    int vertexIndex = id \* 3;

    int posIndex = objectIndices[id] \* 3;

    worldVertices[vertexIndex] = ((modelVertices[vertexIndex] \* rotMatrices[rotateIndex]) + (modelVertices[vertexIndex + 1] \* rotMatrices[rotateIndex + 3]) + (modelVertices[vertexIndex + 2] \* rotMatrices[rotateIndex + 6])) + posVectors[posIndex];

    worldVertices[vertexIndex + 1] = ((modelVertices[vertexIndex] \* rotMatrices[rotateIndex + 1]) + (modelVertices[vertexIndex + 1] \* rotMatrices[rotateIndex + 4]) + (modelVertices[vertexIndex + 2] \* rotMatrices[rotateIndex + 7])) + posVectors[posIndex + 1];

    worldVertices[vertexIndex + 2] = ((modelVertices[vertexIndex] \* rotMatrices[rotateIndex + 2]) + (modelVertices[vertexIndex + 1] \* rotMatrices[rotateIndex + 5]) + (modelVertices[vertexIndex + 2] \* rotMatrices[rotateIndex + 8])) + posVectors[posIndex + 2];

}

\_\_kernel void getRotationMatrix ( //get rotation in matrix format

    \_\_global const float \*rotationVectors,

    \_\_global float \*matrices //This is in column major format (so i can dump it into a matrix3f if i needed to)

)

{

    int vecId = get\_global\_id(0) \* 3;

    int matId = get\_global\_id(0) \* 9; //each matrix is 9 floats. VecId is 3 floats

    float x = rotationVectors[vecId];

    float y = rotationVectors[vecId + 1];

    float z = rotationVectors[vecId + 2];

    matrices[matId] = cos(z) \* cos(y); //from https://en.wikipedia.org/wiki/Rotation\_matrix

    matrices[matId + 1] = sin(z) \* cos(y);

    matrices[matId + 2] = sin(y);

    matrices[matId + 3] = (cos(z) \* sin(y) \* sin(x)) - (sin(z) \* cos(x));

    matrices[matId + 4] = (sin(z) \* sin(y) \* sin(x)) + (cos(z) \* cos(x));

    matrices[matId + 5] = cos(y) \* sin(x);

    matrices[matId + 6] = (cos(z) \* sin(y) \* cos(x)) + (sin(z) \* sin(x));

    matrices[matId + 7] = (sin(z) \* sin(y) \* cos(x)) - (cos(z) \* sin(x));

    matrices[matId + 8] = cos(y) \* cos(x);

}

## net.alevel.asteroids.game

### GameLogic.java

**package** net.alevel.asteroids.game;

**import** java.io.IOException;

**import** java.util.ArrayList;

**import** java.util.List;

**import** org.joml.Vector2f;

**import** org.joml.Vector3f;

**import** org.lwjgl.opengl.GL11;

**import** net.alevel.asteroids.engine.GameEngine;

**import** net.alevel.asteroids.engine.ILogic;

**import** net.alevel.asteroids.engine.Window;

**import** net.alevel.asteroids.engine.graphics.Camera;

**import** net.alevel.asteroids.engine.input.Input;

**import** net.alevel.asteroids.engine.input.enums.NonPrintableChars;

**import** net.alevel.asteroids.engine.input.enums.SpecialChars;

**import** net.alevel.asteroids.engine.objects.GameObject;

**import** net.alevel.asteroids.engine.objects.NonRenderableObject;

**import** net.alevel.asteroids.engine.utils.Pair;

**import** net.alevel.asteroids.game.cl.CLManager;

**import** net.alevel.asteroids.game.noise.Perlin2;

**import** net.alevel.asteroids.game.objects.Asteroid;

**import** net.alevel.asteroids.game.objects.GameObjects;

**import** net.alevel.asteroids.game.objects.ObjectAssembly;

**import** net.alevel.asteroids.game.objects.Ship;

**import** net.alevel.asteroids.game.objects.StaticGameObject;

**import** net.alevel.asteroids.game.objects.shapes.MeshGen;

**import** net.alevel.asteroids.game.physics.Physics;

**import** net.alevel.asteroids.game.physics.RigidObject;

/\*\*Represents the main simulation logic. It manages the simulation and tells it what to do and what to render each time the game loop asks.

\* The functions here are called every time the game loop in {@link GameEngine} repeats

\*/

**public** **class** GameLogic **implements** ILogic {

**public** **static** **final** **float** ***CAMERA\_POS\_STEP*** = 0.01f;

**public** **static** **final** **float** ***MOUSE\_SENSITIVITY*** = 0.05f;

**private** **final** Camera camera;

**private** **final** GameObjects gameObjects;

//private final List<GameObject> gameObjects;

**private** Ship player;

**private** RigidObject[] rigidObjects;

**private** **final** List<ObjectAssembly> ships; //AI ships

**private** **final** Physics physics;

**private** GameLogic() **throws** IOException {

CLManager.*init*();

**this**.camera = **new** Camera();

**this**.gameObjects = **new** GameObjects(); // ArrayList<GameObject>();

**this**.ships = **new** ArrayList<ObjectAssembly>();

**this**.physics = **new** Physics();

}

@Override

**public** **void** init(Window window) **throws** Exception {

System.***out***.println(GL11.*glGetString*(GL11.***GL\_VERSION***));

//SATCL.init();

//MatrixMulTest.run();

//SurfaceNormalsTest.run();

//SurfaceNormalsTest.runJava();

//MatrixVectorMulTest.run();

//GetMinMaxPoints.run();

**this**.player = **new** Ship();

**this**.player.rotate(0, 90, 0);

**this**.rigidObjects = **new** RigidObject[] {

**new** RigidObject(MeshGen.*cube*(1, 1, 1)),

**new** RigidObject(MeshGen.*cube*(1, 1, 1))

};

**this**.rigidObjects[1].setPosition(-20, 0, 0);

**this**.rigidObjects[0].setPosition(-20, 0, -10);

**this**.gameObjects.spawnAll(**this**.rigidObjects);

**this**.gameObjects.spawnObject(Perlin2.*example*());

StaticGameObject asteroidObject = **new** Asteroid();

asteroidObject.setPosition(0, 60, 0);

**this**.gameObjects.spawnObject(asteroidObject);

//Grid.debug(this.gameObjects);

//this.gameObjects.spawnObject(new StaticGameObject(new Mesh(asteroid.getPositions(), new float[0], new float[0], asteroid.getIndices())));

**this**.gameObjects.spawnObject(**this**.player);

//this.gameObjects.add(new StaticGameObject(MeshGen.triangularPrism(new Vector2f(0, 0), new Vector2f(1, 0), new Vector2f(0, 1), 10)));

//this.gameObjects.add(new StaticGameObject(MeshGen.sphere(2)));

}

@Override

**public** **void** update(**float** accumulatedTime, **float** interval, Input input) **throws** IOException {

**final** Vector3f cameraInc = **new** Vector3f();

**if**(input.isKeyPressed('W') && !input.isKeyPressed('S'))

cameraInc.z = -1f;

**else** **if**(input.isKeyPressed('S') && !input.isKeyPressed('W'))

cameraInc.z = 1f;

**if**(input.isKeyPressed('A') && !input.isKeyPressed('D'))

cameraInc.x = -1f;

**else** **if**(input.isKeyPressed('D') && !input.isKeyPressed('A'))

cameraInc.x = 1f;

**if**(input.isKeyPressed(NonPrintableChars.***LEFT\_SHIFT***) && !input.isKeyPressed(SpecialChars.***SPACE***))

cameraInc.y = -1f;

**if**(input.isKeyPressed(SpecialChars.***SPACE***) && !input.isKeyPressed(NonPrintableChars.***LEFT\_SHIFT***))

cameraInc.y = 1f;

**float** posStep;

**if**(input.isKeyPressed(NonPrintableChars.***LEFT\_CTRL***))

posStep = 100f;

**else**

posStep = 20f;

camera.movePosition(cameraInc.mul(***CAMERA\_POS\_STEP***).mul(posStep));

**this**.player.translate(cameraInc);

//this.player.translate(cameraInc.x \* CAMERA\_POS\_STEP \* posStep, cameraInc.y \* CAMERA\_POS\_STEP \* posStep, cameraInc.z \* CAMERA\_POS\_STEP \* posStep); //player always in same position as camera

Vector2f rotVec = input.getDeltaMousePos().mul(***MOUSE\_SENSITIVITY***);

camera.moveRotation(rotVec.x, rotVec.y, 0);

**this**.player.rotate(rotVec.x \* -1, rotVec.y \* -1, 0);

//this.rigidObjects[0].translate(0, 0, -.01f).rotate(0, (float) Math.PI / 3, 0);

**this**.rigidObjects[0].translate(0, 0, .01f);

**this**.physics.onUpdate(**this**.gameObjects.getRigidObjects());

List<NonRenderableObject> objects = **this**.gameObjects.getAllObjects();

**for**(**int** i = 0; i < objects.size(); i++)

objects.get(i).update(accumulatedTime);

**for**(**int** i = 0; i < objects.size(); i++)

objects.get(i).onUpdateFinish(interval);

}

@Override

**public** Pair<Camera, List<GameObject>> toRender() {

**return** **new** Pair<Camera, List<GameObject>>(**this**.camera, **this**.gameObjects.getRenderableObjects());

}

@Override

**public** **void** cleanUp() {

**for**(NonRenderableObject o : gameObjects.getAllObjects())

o.cleanUp();

//SATCL.cleanUp();

//WorldCoords.cleanUp();

}

**public** **void** addObject(GameObject o) {

**this**.gameObjects.spawnObject(o);

}

**public** **void** removeObject(GameObject o) {

**this**.gameObjects.despawnObject(o);

}

**private** **static** GameLogic *instance*;

**public** **static** GameLogic init() **throws** IOException {

**if**(*instance* == **null**)

*instance* = **new** GameLogic();

**return** *instance*;

}

**public** **static** GameLogic getInstance() {

**return** *instance*;

}

}

/\*AABBf aabb = aabbTest.getBoundingBox();

float[] boundingBoxVertices = {

aabb.maxX, aabb.maxY, aabb.maxZ,

aabb.minX, aabb.minY, aabb.minZ,

aabb.maxX, aabb.maxY, aabb.minZ,

aabb.minX, aabb.maxY, aabb.minZ,

aabb.minX, aabb.maxY, aabb.maxZ,

aabb.maxX, aabb.minY, aabb.minZ,

aabb.maxX, aabb.minY, aabb.maxZ,

aabb.minX, aabb.minY, aabb.maxZ,

};

//System.out.println(aabbTest.getModelBoundingBox());

//System.out.println(aabb);

//System.out.println(aabbTest.getPosition() + " | " + aabbTest.getRotation() + " | " + aabbTest.getScale() + " | " + aabbTest.getBoundingBox());

//System.out.println(Arrays.toString(boundingBoxVertices));

int[] indices = {

0, 2, 3,

0, 4, 3,

2, 5, 1,

2, 3, 1,

2, 5, 6,

2, 0, 6,

3, 4, 1,

4, 7, 1,

0, 6, 7,

0, 4, 7,

5, 6, 7,

5, 1, 7,

};

this.gameObjects.remove(this.boundingBox);

this.boundingBox = new StaticGameObject(new Mesh(boundingBoxVertices, boundingBoxVertices, boundingBoxVertices, indices));

this.gameObjects.add(this.boundingBox);\*/

/\*Mesh mesh = WavefrontMeshLoader.loadMesh("/models/bunny.obj");

mesh.setColour(new Vector3f(0f, 1f, 0f));

GameObject o = new StaticGameObject(mesh);

o.setScale(1.5f);

o.setPosition(0, 0, -2);

Mesh mesh1 = WavefrontMeshLoader.loadMesh("/models/bunny.obj");

mesh1.setColour(new Vector3f(0f, 1f, 0f));

GameObject o1 = new StaticGameObject(mesh1);

o1.setScale(1.5f);

o1.setPosition(0, 1, -2);

Mesh mesh2 = WavefrontMeshLoader.loadMesh("/models/bunny.obj");

mesh2.setColour(new Vector3f(0f, 1f, 0f));

GameObject o2 = new StaticGameObject(mesh2);

o2.setScale(1.5f);

o2.setPosition(0, -1, -2);

this.gameObjects.add(o);

this.gameObjects.add(o1);

this.gameObjects.add(o2);\*/

/\*this.gameObjects = new GameObject[8];

for(int i = 0; i < 6; i++)

this.gameObjects[i] = new StaticGameObject(WavefrontMeshLoader.loadMesh("/models/bunny.obj"));

this.gameObjects.setPosition(0, 0, -1).getMesh().setColour(new Vector3f(0, 1, 0));

this.gameObjects[1].setPosition(0, 0, 1).setRotation(0f, 180f, 0f).getMesh().setColour(new Vector3f(1, 0, 0));

this.gameObjects[2].setPosition(0, -1, 0).getMesh().setColour(new Vector3f(0, 0, 1));

this.gameObjects[3].setPosition(0, 1, 0).getMesh().setColour(new Vector3f(0, 1, 1));

this.gameObjects[4].setPosition(-1, 0, 0).getMesh().setColour(new Vector3f(1, 1, 0));

this.gameObjects[5].setPosition(1, 0, 0).getMesh().setColour(new Vector3f(1, 0, 1));

//this.gameObjects[6] = new Projectile(WavefrontMeshLoader.loadMesh("/models/bunny.obj"));

//this.gameObjects[6].setPosition(0, 1, 1).getMesh().setColour(new Vector3f(1, 0.5f, 0));

this.gameObjects[6] = new Projectile(WavefrontMeshLoader.loadMesh("/models/cube.obj"));

((Projectile) this.gameObjects[6]).setHorizontalAngleProjected(270).setPosition(0, 0, 0).setScale(0.05f);

this.gameObjects[7] = new PhysicalObject(WavefrontMeshLoader.loadMesh("/models/cube.obj"));

this.gameObjects[7].setPosition(0, 0, -1).setScale(0.1f);\*/

/\*this.tempProjectile = new Projectile(WavefrontMeshLoader.loadMesh("/models/cube.obj"));

this.tempProjectile.setHorizontalAngleProjected(270).setPosition(0, 0, 0).setScale(0.05f);

this.gameObjects.add(this.tempProjectile);

this.tempPhysicalObject = new PhysicalObject(WavefrontMeshLoader.loadMesh("/models/cube.obj"));

this.tempPhysicalObject.setPosition(0, 0, -1).setScale(0.1f);

this.gameObjects.add(this.tempPhysicalObject);\*/

### Main.java

**package** net.alevel.asteroids.game;

**import** java.io.IOException;

**import** net.alevel.asteroids.engine.GameEngine;

/\*\*The entry point of the program. It doesn't do much since {@link GameLogic} handles all the main logic

\*/

**public** **class** Main {

**public** **static** **void** main(String... args) **throws** IOException {

**new** GameEngine(GameLogic.*init*()).run();

}

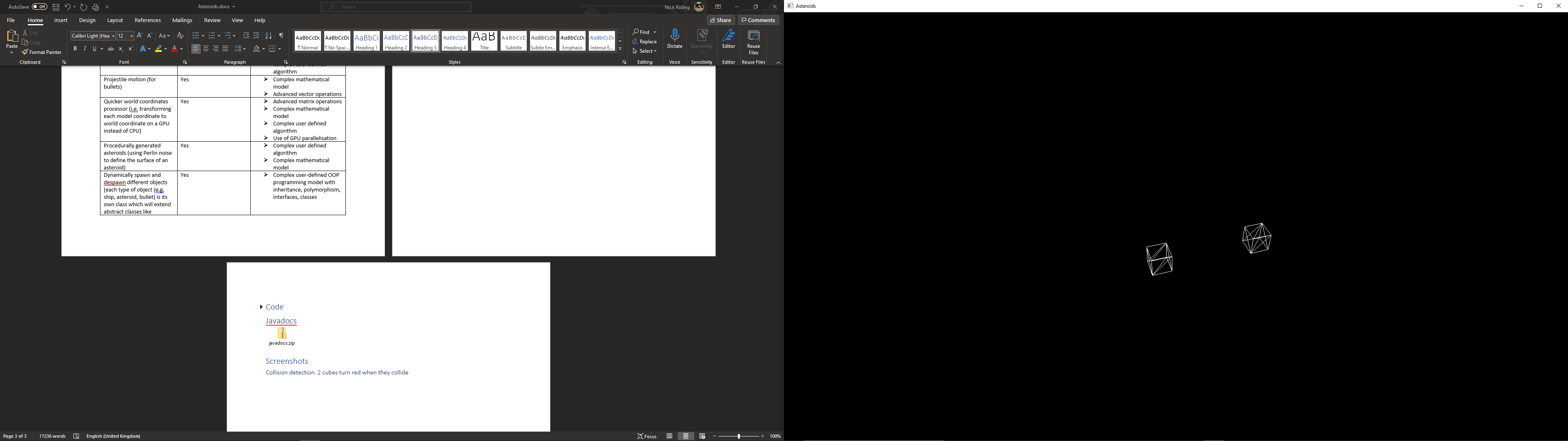
}

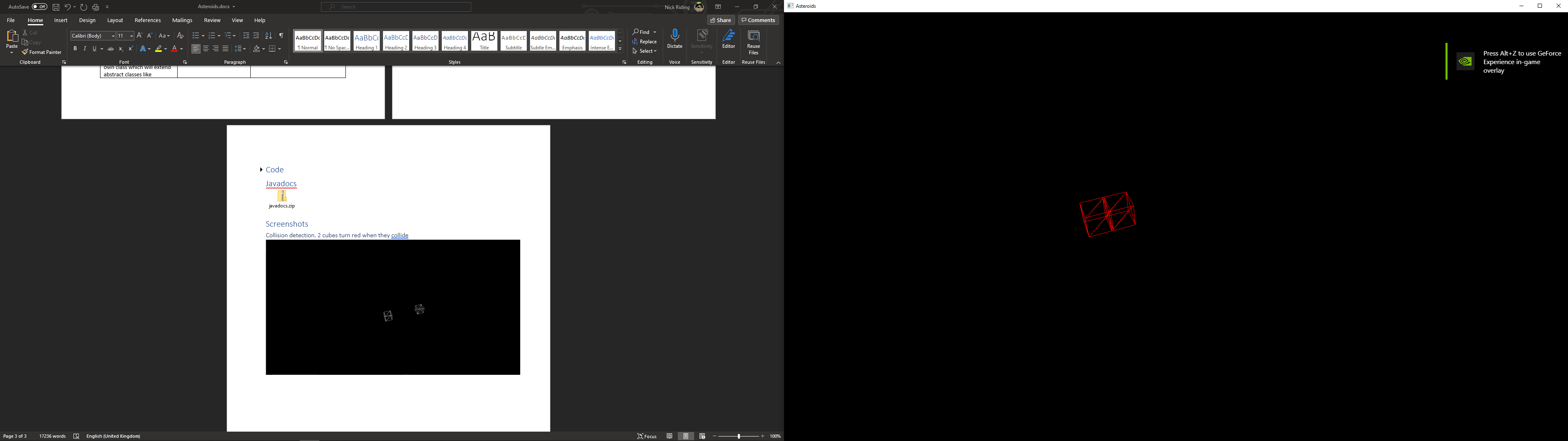
# Javadocs



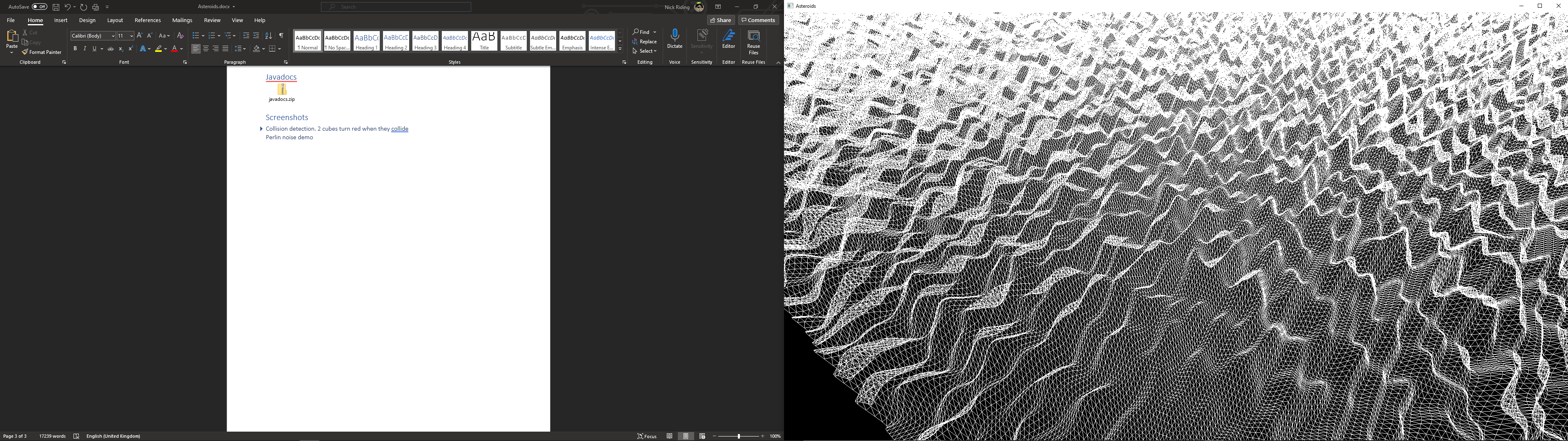
# Screenshots

### Collision detection. 2 cubes turn red when they collide:

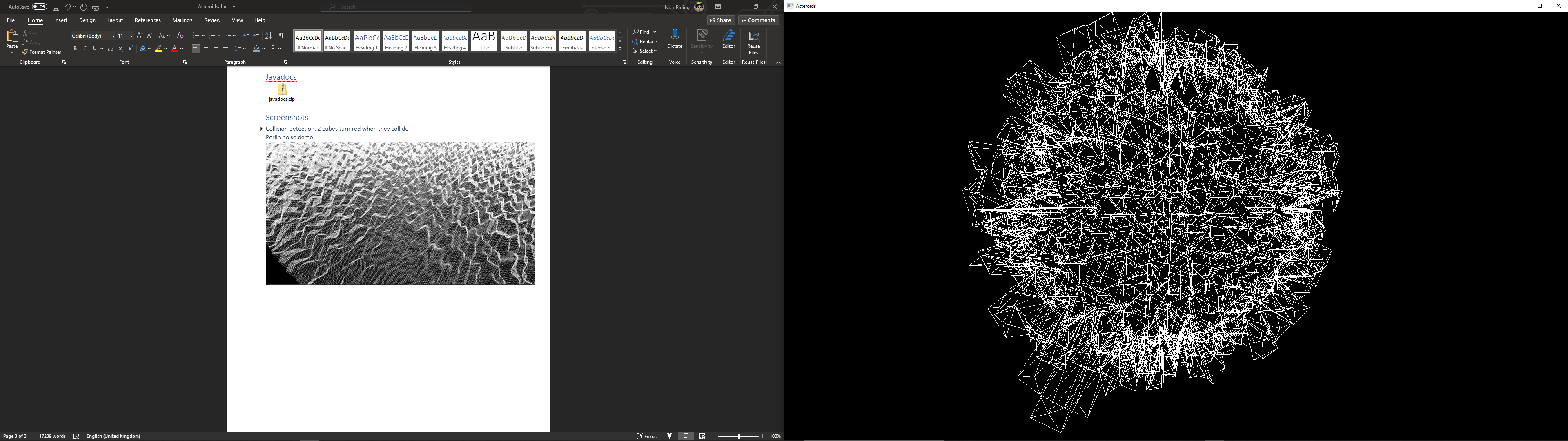




### Perlin noise demo:



The image above is a height map that is wrapped around a sphere to create an asteroid:



### Object assembly example:

This screenshot shows how I made a ship using basic regular 3D shapes:

