Lecks Chester

11254915

ITC320 – ADVANCED GRAPHICS FOR GAMES

Assignment 1

Contents

[Instructions 1](#_Toc522312944)

[Introduction 1](#_Toc522312945)

[Structure 1](#_Toc522312946)

[Classes 1](#_Toc522312947)

[Game 2](#_Toc522312948)

[Mesh 2](#_Toc522312949)

[MeshPart 2](#_Toc522312950)

[Shader 2](#_Toc522312951)

[Car 2](#_Toc522312952)

[Cameras 2](#_Toc522312953)

[BoundingBox 3](#_Toc522312954)

[Shaders 3](#_Toc522312955)

[Changes to parsers and Common folder 3](#_Toc522312956)

[Car Physics 4](#_Toc522312957)

# Instructions

Unzip the package, run serverV3\_revised.py and open <http://localhost:8080/Assignment1.html> .

If performance is an issue, Game.buildingLayers can be reduced.

To see the difference, the following line in Game.init() can be commented to disable hardware instancing:

this.gl.ANGLE\_instanced\_arrays = this.gl.getExtension("ANGLE\_instanced\_arrays");

# Introduction

Originally, I was planning on trying to make a decent city generation algorithm, but after seeing the art we could use, I decided to concentrate more on the car. To make the result more interesting, I have made the decision to deviate from the instructions in a couple of ways:

* The car drives around a block of buildings instead of a single building.
* The car fades to the different colours. It still changes between red, green and blue every 3 seconds, but it fades between them.

The game is contained in a global variable ‘game’ which is an instance of the Game class. This global variable is only used for instrumentation (counting polygons, etc). Other pollution in the global namespace includes all of the class names (Mesh, MeshPart, Car, Shader, ChaseCamera, etc.) and some math function names.

The code is documented with jsdoc (and referenced where appropriate).

# Structure

## Classes

The app is divided up into the following classes:

### Game

The main class. Loads the models and sets up instances, receives/passes on input and handles the canvas, timing, and main game loop. The name ‘game’ is meant to represent a generic reusable name for an application of this type (usually games).

### Mesh

The mesh class I used was initially written for ITC363. It supports some features that aren’t used in this assignment such as normals and vertex colors, but I figured I would leave them in so that I can re-use the class again.

The class that loads and stores data for meshes. Can be used either with procedural meshes (none of which are used in this assignment). The data stored includes the position, normal, color, uv and index arrays and buffer Ids, as well as a list of MeshParts (described below).

The class also includes a modified version of the ObjParser that works asynchronously.

The class supports hardware instancing, meaning instead of storing matrices in uniform values, they are stored in attributes that can be passed in as arrays. Hardware instancing is automatically enabled if getting the required WebGL1 extension (ANGLE\_instanced\_arrays) was successful. This did create some complications. While GLSL supports mat4 attributes, WebGL only supports vec4. To work around this, the 4 vec4s that make up the matrix had to be bound to the array (or set when instancing is disabled) separately. In most cases, I would use a lot more compact data for this purpose, but the assignment specifically asked for translation, rotation and scale to occur in the vertex shader, so I’m using the entire world matrix as instance data.

### MeshPart

MeshParts contain the name of the material for the part, the index offset and vertex count for the part, and the Shader for the part. No actual buffers are in the MeshPart; they are in the containing Mesh.

### Shader

The Shader class contains a reference to an actual shader program, and references to all the attribute and uniform values in the shader. It also contains some helper functions for the instancing described above.

### Car

The Car class contains all of the meshes that make up a car, and all of the properties for that car. If there was more than 1 car, this would likely have to be structured a little differently (the meshes would be in a separate class).

Once the car Mesh is loaded, several copies of it are made to represent wheels and brakes. All copies contain the same references to buffers (so there’s not a position buffer for each part for example). The parts of the model that are drawn for each Mesh is adjusted by giving them the correct MeshParts. The car paint shader is applied to the mesh parts that are painted.

### Cameras

All of the cameras contain a name, a view projection matrix, methods to handle movement and rotation input, and a method to update the matrix. Normally I would have an abstract base class or interface, but that’s not an option that I know of in Javascript.

### BoundingBox

A simple bounding box class that stores the min and max components of vectors that have been passed in and calculates the size and centre of the bounding box.

# Shaders

There’s not a lot to say about the shaders. A single vertex shader is used for everything, and simply transforms the position and passes on the texture coordinates. There are 3 fragment shaders. One is a standard textured fragment shader. The ground fragment shader combines a highly tiled copy of the texture with a stretched-out copy to achieve a less repetitive look. The car paint fragment shader interpolates between the standard car texture sample, and versions with the red channel swapped with the green/blue channel.

In order to parts of the shaders sharable (there is no #include in WebGL), I left the shaders in JS files.

The scripts are referenced in Game.GLSL.vsName or Game.GLSL.fsName.

# Changes to parsers and Common folder

I made a single change made to the Common folder in MV.js’s flatten method to make it properly handle nested arrays and arrays of matrices. The application will not work without this modified flatten method.

The parsers were modified to download asynchronously. The way this basically works is:

1. The mesh is initially constructed with blank arrays/buffers. The downloadObj method is then called, which sends a http request to retrieve the OBJ file and the program continues running. A callback can be passed to the downloadObj function to run when the mesh is entirely loaded (called the ‘then’ function). This allows any processing that requires the mesh data and/or materials to take place when appropriate.
2. The browser will likely draw some frames. Any instances of the mesh will technically draw, but nothing will be visible because their buffers are empty.
3. When the OBJ file finishes downloading, the loadObj method runs which load the mesh data into the arrays. If there were no MTL files specified in the OBJ file, the ‘then’ function is called and the process is complete. Otherwise, downloadMtl is called and passed the ‘then’ function, which begins a http request to retrieve the MTL file.
4. The browser will likely draw some frames. The buffers will be updated on the first, and the mesh will draw with the default material (black).
5. Upon receiving the MTL file, the applyMtl callback uses a slightly modified MtlParser (that takes the code instead of filename) to apply the materials to the MeshParts. Any textures specified in the MTL are created as blank textures have http requests sent to retrieve them (using a modified version of the TGAParser, which caches textures and downloads asynchronously). The process of calling downloadMtl to start downloads and calling back to loadMtl continues until all Mtl files have been processed, at which point the ‘then’ function runs.
6. The browser will likely draw some frames. Any textures that haven’t finished downloading will show as blue.
7. The texture HTTP requests finish and the TGA files are decoded and sent to the texture buffer.

# Car Physics

The car has spinning/turning wheels and fake physics. None of this is realistic; only a visual simulation. The “physics” includes a tilt and a sliding/drifting effect.

