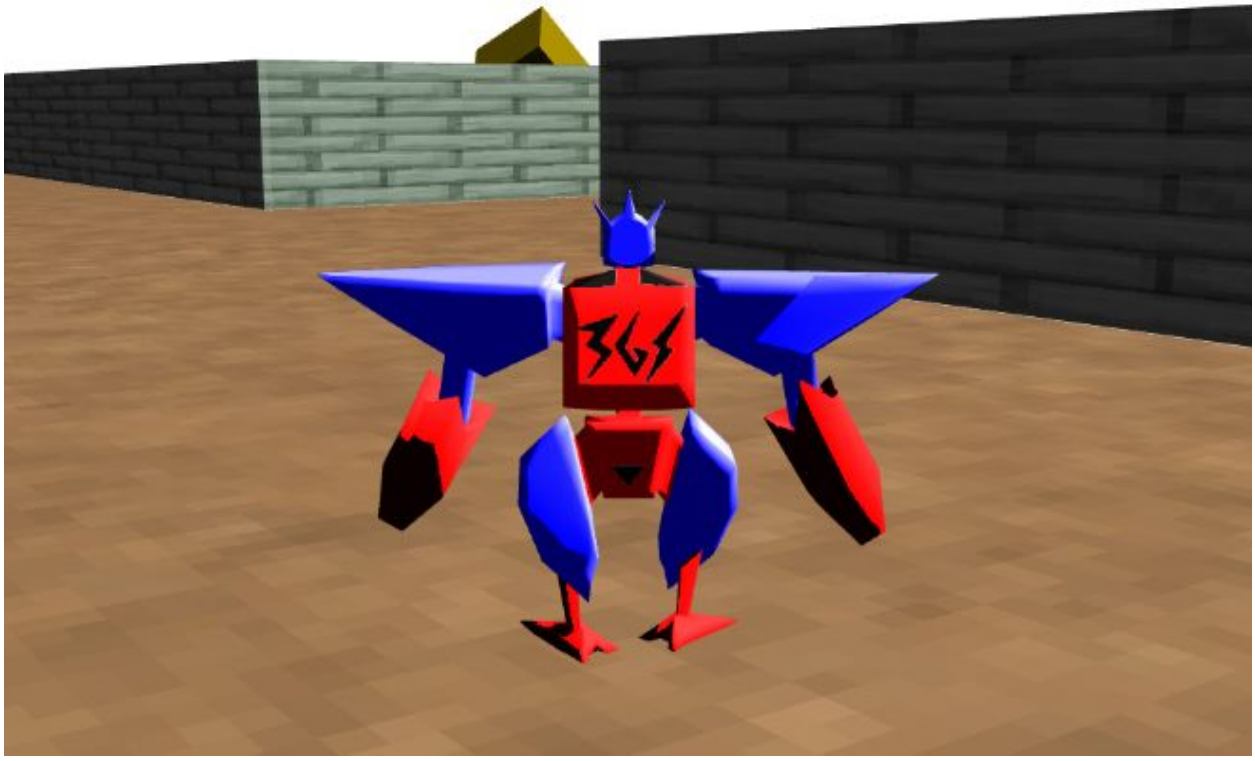
3GS

A maze exploration game built with WebGL



Introduction

This project features a playable maze exploration game, where the player controls a humanoid mech that has to traverse a procedurally generated maze in order to reach a trophy. The player can choose the size of the maze to explore, and he can use the WAD keys to move, as well as the mouse scroll wheel to control the height level of the camera.

We decided, both as an exercise in software engineering and as a way to have something we could eventually reuse and expand later on, to approach the development of the game

by splitting our software in two: a library and a game. The library is a minimal game engine, abstracting basic WebGL functions and implementing some utilities so that a simple game can easily be created using it. The game is basically a tech demo using the library, which is completely decoupled from it.

Game logic

Maze generation

The mazes are generated using a custom implementation of the modified Prim's algorithm. This algorithm starts with a grid full of walls and recursively explores all the grid cells at random, removing walls when possible to connect adjacent grids and create a path that the player can explore. All the logic for the maze generation is completely decoupled from the rest of the graphics-related code, including the code that is responsible for creating the actual maze geometry from the maze grid structure. This grid is used to check for collisions while the player is moving as well as to check when the player has reached the target cell. The geometry is built by simply iterating over all the cells in the maze, and constructing new cubes as needed, and then placing them in the right coordinates in the world space. The same is done to also place the player and target cell, which are spawned by the maze generation code in specific coordinates, and then rendered on screen with the rest of the geometry.

Collision detection

The collision detection is performed on a 2D plane only, taking into account the current maze structure and the player position. This greatly reduces the number of required operations, but it's sufficient in this case as the movements of the player are constrained on the ground plane. In order to check for collisions, whenever the player moves the game checks a number of coordinates along the circumference of a circle that is centered on the next position the player would be moved to. If any of those positions is inside any of the cubes that make up the maze, the movement is canceled. This makes it so that the player is blocked within a given margin to each game object, so that geometry interpenetrations are also avoided, since the collision checks are only computed with respect to the coordinate at

the center of the player model, without taking into account the exact geometry of the player's model.

Rendering

Perhaps, the first thing our library aimed to abstract was the whole process of calling the WebGL library and writing shaders. Our **GraphicObject**, **Scene** and **Light** classes cooperate to completely hide this process from the user, along with the pre-made shaders in the **shaders** folder.

The workflow to create a 3D scene and render an image from it, which we used in our game, is as simple as this:

1. Set up an HTML5 canvas.
2. Create a new 3GS **Scene** with that canvas as an argument.
3. Add a 3GS **Light** and a 3GS **Camera** to the Scene.
4. Add some objects to the scene, either with the **Cube** primitive we offer, or importing them from an obj file with the obj importer, and position them as described in the GraphicObject subsection.
5. Fire the render method on the Scene.

Graphic Object

The GraphicObject class implements a tree-based hierarchical object. Each component is a GraphicObject instance, that has its own buffers with vertices, normals and uvCoords. It can have children, that are the components of the next level in the hierarchy. The render function traverses the tree starting from the root, pushing at each level the instance matrix of the parent, in order to position each component in the right place with respect to the parent. In addition, this class implements methods (rotate, translate, scale) to transform each component with respect to the origin of the axes (object coordinates). We decided to separate the logic of the hierarchical object, including the rendering and the geometric transformations, from the geometry itself. This is to facilitate the import of objects with complex geometry. An example is the main character, whose geometry is modelled using

Blender, and successively it's assembled into a hierarchical object by the obj importer that will be described later on. The geometric information that needs to be passed to WebGL is stored in three arrays, collecting the vertices, normals and UV coordinates. At render time this info is bound to the GPU buffers so that the object can be properly rendered.

```
// binding vertex buffer
this.scene.gl.bindBuffer(this.scene.gl.ARRAY_BUFFER, this.vBuffer);
this.scene.gl.vertexAttribPointer(this.scene.gl.getAttribLocation(this.scene.program,
"vPosition"), 4, this.scene.gl.FLOAT, false, 0, 0);
this.scene.gl.enableVertexAttribArray(this.scene.gl.getAttribLocation(this.scene.progr
am, "vPosition"));
```

Every object also has its instance matrix, which is updated without the user needing to know every time the object is translated, rotated or scaled. The matrix contains the relative transformations to the parent object, if there is one, and the parent's model matrix is passed as a parameter when rendering, so that it can be multiplied to the object's to obtain its own model matrix. If the object is directly a child of the scene, an identity matrix will be passed to him as a parameter instead, as its own instance matrix will already be the required model matrix.

```
render(parentMatrix) {
…
var modelMatrix = mult(parentMatrix,this._instanceMatrix);
…
}
…
translate(x, y, z) {
        this._pos[0] = this._pos[0] + x;
        this._pos[1] = this._pos[1] + y;
        this._pos[2] = this._pos[2] + z;
        this._instanceMatrix = mult(translate(x, y, z), this._instanceMatrix);
    }
```

We also defined the useful geometric primitive "Cube" that extends GraphicObject. The class Cube, as an extension of the GraphicObject, also includes the geometry (vertices, normals, uvCoords and color). We used this primitive to build the maze. This choice is due to the fact that it was easier and way more suited to the procedural nature of the maze to use multiple cubes to transpose the 2D matrix representation of the maze into a renderable object, than modelling the geometry of the maze directly.

Scene

The Scene object is the most important element of 3GS: everything happens through the scene, as it contains the WebGL context, the program (created by importing the shaders from text files), the camera, the light and references to all the 3D objects and animations. When the renderScene method is called on the scene, the render methods are called on the first level children Graphic Objects, and those will recursively call render on their children. The animations are also managed through the scene's renderScene method, but this will be better described in their own section. Every object in the scene has a reference to the scene set, since it's so important. This happens either when the object is manually added to the scene as a first level child, or when an object is set as child of another object belonging to the scene.

```
renderScene(){
        this._gl.clear( this._gl.COLOR_BUFFER_BIT | this._gl.DEPTH_BUFFER_BIT);
        var currViewMatrix = this._activeCamera.getViewMatrix();
        // Update the view matrix on the GPU
        this._gl.uniformMatrix4fv(this._gl.getUniformLocation(this.program,
"viewMatrix" ), false, flatten(currViewMatrix));
        var currProjMatrix = this._activeCamera.getProjectionMatrix();
        // Update the projection matrix on the GPU
        this._gl.uniformMatrix4fv(this._gl.getUniformLocation(this.program,
"projectionMatrix" ), false, flatten(currProjMatrix));
        if(this._animations != null && (this._animations.length != 0))
            //Updates scene objects animation parameters
            this._animateScene();
        var len = this._objects.length;
        if(this._objects != null && (len != 0)){
            for(var i = 0; i < len; i++){
                if(this._objects[i] != null)
                    //renders a GraphicObject
                    this._objects[i].render(new mat4());
            }
        }
    }
```

Light

There is not that much to say about the lights. For simplicity, we decided to only implement a Directional Light, and to only support one light in the scene (since our game only has the

light of the sun from above), which also has an ambient component to simulate the reflection of the sun rays lighting up even the shadows a little bit. The light has a direction (which is a vec4 with the 4th value set to 0), an ambient, a diffuse, and a specular value, all of which are used by the fragment shader to compute lighting.

Camera

The role of the camera is basically to give to the scene, when rendering, the view matrix and the projection matrix. The scene sets them once for frame on the GPU as they're the same for all the objects. These matrices are computed by the Camera object using the eye, at, up, fovY, aspect, near and far attributes which are set by the user. Since our game only uses a perspective camera, we only implemented the PerspectiveCamera class, which relies on the MV.js perspective and lookAt functions.

.obj Importer

Our library includes also an importer for the format obj. The obj file extension is an open format used to represent the geometry of a 3D object. It's widely used in commercial graphic applications. A file with this format is organized as follows:

- A line starting with "o" followed by the name of the component (o Torso)
- The list of vertices, each line starting with the letter "v" followed by the x,y,x (and optional w) coordinates (v 0.404199 0.664359 0.477320)
- The list of uv coordinates, each line starting with "vt" followed by u,v coordinates (vt 0.689384 0.615870). UV coordinates are used to map a 2D image onto the 3D object.
- The list of faces. Each line starting with "f" defines one of the polygons in which the object is subdivided. A polygon can have an arbitrary number of vertices (it could be a triangle, quadrilateral and so on), however the objParser will further decompose polygons with more than 3 vertices into triangles, since most graphic cards support only triangles. Each face is identified by n components, where n is the number of vertices of the face, and each component has three terms separated by the "/" character. The first term is the index of the vertex, the second is the index of the normal and the third is the index of the uv coordinates that must be mapped onto that vertex. For example the line "f 61/1/1 2/2/2 10/3/3 62/4/4" represents a polygon with four vertices (quadrilateral) a,b,c,d. The coordinates of the vertex "a" are at

position 61 in the vertices list, the coordinates of the normal of the vertex “a” are at position 1 in the normals list, while the uv coordinates to be mapped on vertex “a” are at position 1 in the uv coordinates list.

The “Parse” class takes as input the URL of the .obj file and produces for each component the array of vertices, normals and uv coordinates.

Shaders

The vertex and fragment shaders are stored, respectively, in a phong.vert and a phong.frag file, as they are inspired to the Phong reflection model. I said inspired because, rather than aiming to achieve perfect adherence to the model, we tweaked the parameters in order to have sharp contrasts with shadows and vivid colors, to recreate a style reminescent both of the works of the animation studios Gainax and Trigger (especially Gurren Lagan) and of cel-shaded games that were really popular in the early 2000s.

The vertex shader computes the position for each vertex by post-multiplying its position to the projection, view and model matrix. It also computes the normal for the vertex in a similar fashion, the vertex going from the vertex to the light and the one going from the vertex to the viewer’s position, as those will need to be interpolated by the rasterizer and passed to the fragment shader. The texture coordinates are also interpolated.

```
vec3 pos = (viewMatrix * modelMatrix * vPosition).xyz;
    vec3 light = (viewMatrix * lightPosition).xyz;

    L = light - pos;
    E = -pos;
    N = inverseTransposedModelMatrix * vNormal.xyz;

    fTexCoord = vTexCoord;

    gl_Position = projectionMatrix * viewMatrix * modelMatrix * vPosition;
```

The fragment shader normalizes the now interpolated vectors, as normalizing them in the vertex shader would not have guaranteed that, after the interpolation, they would have the unit magnitude we need in the lighting calculation. Then, for each fragment, the exact color

is computed as a sum of the ambient, specular and diffuse components, all of which are calculated on the basis of the interpolated values. In case the object has a texture, the value of the fragment's color is also based on the texture's values at the interpolated coordinates.

Animations

3GS also provides some useful APIs with the purpose of "giving life" to every object contained into a scene. These APIs are organized in two specific classes, the **KeyframeShift** class and the **Animation** class. Let's take a more detailed look.

KeyframeShift

This class is a simple implementation of the **keyframe** concept. It contains the set of **parameters** of a single component of a GraphicObject that we want to change over time, like the **position**, **rotation** angle and **scale factor**.

Each KeyframeShift object is initialized with both the initial and the final values of the parameters we want to animate. The in between values are computed through **linear interpolation** with a customizable number of **steps.** At each render cycle the update() function computes the current values of the parameters and calls the setter methods to reflect those changing in the object position, rotation and scaling. To implement animation loops we need to repeat the same KeyframeShift multiple times. To do that all the parameters must be set to their initial values and those changes must be reflected on the object. This task is performed by the reset() method.

Animation

Once we have built enough KeyframeShift objects, they have to be passed to an instance of the **Animation** class as a Javascript **Array**. You also need to specify whether it's a loop or not with the **isLoop** parameter.

The Animation class is in charge of **sequentially processing** the KeyframeShift array. If the animation is **not a loop**, then every KeyframeShift is **removed from the array** right after it has reached its end.

The more Animation objects are pushed into a scene, the more transformations over multiple objects are performed at the same time. That is, at each **render()** call, the **first element of each Animation object** belonging to the scene is **processed**. So those KeyframeShift transformations are applied at the same time.

Finally, the class exposes **play()** (default animation state) and **stop()** methods.

Controllers

The ObjectController class allows the user to control the object position in the xz plane. This is done by registering handlers to the events of pressing and releasing the “w-a-d” buttons on the keyboard. The forward movement is implemented by translating the object in the direction pointed by the “current direction” vector which length is fixed. The rotation performs two steps. The first consists in rotating the “current direction” vector, so that the next forward movement will be affected by the rotation. In the second step the object is rotated by the same angle of the “current direction” vector so that the front of the object will always face the same direction of the “current direction” vector. To rotate the object w.r.t its center we first need to go back to the object coordinates. This can be achieved by translating the object to the origin before computing the rotation, and then translating it back after the rotation.