# COMP3811: Computer Graphics
# Coursework 2
# Interactive Animated Scenes with OpenGL

Joe Jeffcock
sid: 201172812
sc18j3j

January 7, 2021

## Contents

# 1 Introduction

Horror films from the mid-to-late 20th century have a unique atmosphere due in part to their use of shadows. The stark contrast between lit and unlit surfaces allowed for dramatic effects that involved the manipulation of light sources to generate interesting shadows.

Working with lighting in OpenGL, I was immediately inspired by Alfred Hitchcock's Psycho (1960) to create an atmospheric 3D scene with heavily contrasting shadows and moving light sources.

The scene would make use of OpenGL to present dramatic texturing and lighting of objects in the scene, and feature various animated instances of oscillation to create a visually interesting atmosphere. Complex geometry required would also present an opportunity to learn skills in 3D modelling software, a personal and professional goal of mine towards my interests in art and robotics.

## 1.1 Aims

- Meet the requirements of each band of the project specification.
- Learn how to create convex objects from polygons using OpenGL and 3DS Max.
- Create a scene with the visual atmosphere of a 20th century horror film.

## 1.2 Context

To better contextualise the report of this project, the final result of the visual scene is given by Figure 1. This representation will hopefully give motivation to some of the design choices that may not be easily conveyed by the drafts and graph plots created during the initial phases of development.

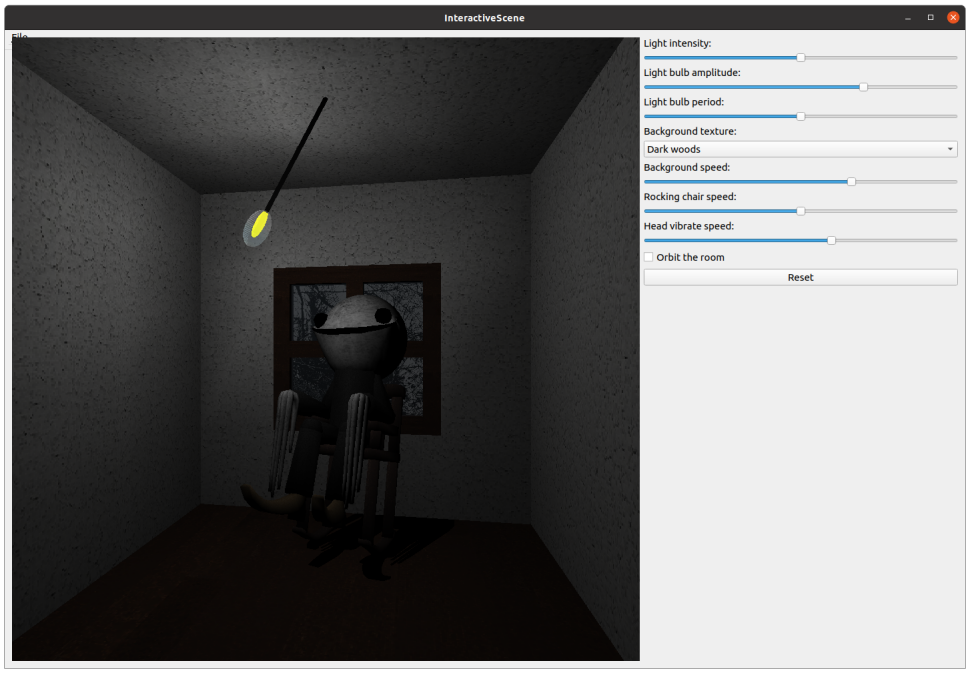


Figure 1: Screenshot of the final product

# 2 Band 4 (40%-50%)

The requirements of Band 4 are as follows:

1. a visual scene demonstrating reasonable complexity through instancing.
2. light and material properties that highlight specular and diffusive light contributions.

A rough layout of the scene was drawn beforehand to plan for these requirements:

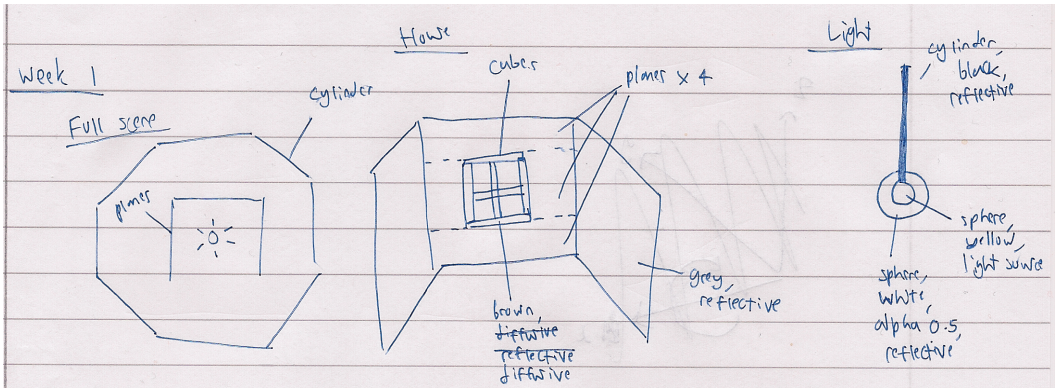


Figure 2: Draft of the complex scene

## 2.1 Basic objects

To facilitate the instantiation of objects in populating our scene, a number of basic shapes were defined and implemented as described in Table 1.

| Shape | Implementation | Orientation | Description |
|---|---|---|---|
| square | GL_POLYGON | outside | tetragon from (0,0,0) to (1,1,0). |
| cube | GL_POLYGON | outside | 9 outward-facing squares forming a cube. |
| cylinder | GL_POLYGON | inside | taken from COMP3811 tutorials.<br>normals were negated to face inwards. |
| sphere | glut object | inside | gluSphere with quadric orientation GL_INSIDE |

Table 1: Basic shapes

The choice of shapes was based on the draft in Figure 2, where cylinders and spheres have their normals pointing inwards as they were designed only to contain light sources based on the initial plan. The function definitions and the implementation of the square shape can be found in Figures 3 and 4 respectively.

## 2.2 Complex scene

The shapes defined in Table 1 were successfully instantiated in the scene according to the draft in Figure 2 as illustrated in Figures 5 to 8. Materials defined in Figure 9 of type `materialStruct` (Figure 3) were used to accentuate the specular and diffusive lighting provided by OpenGL.

The code generating the layout of objects in the scene involves a significant number of transformations that may be hard to visualise, so the five main segments are described in this section.

```c
#ifndef __SC18J3J_SHAPES__
#define __SC18J3J_SHAPES__

#include <GL/glu.h>
#include <GL/glut.h>


/** Material
 * struct comprising phong
 * lighting values.
 **/
typedef struct materialStruct {
    GLfloat ambient[4];
    GLfloat diffuse[4];
    GLfloat specular[4];
    GLfloat shininess;
} materialStruct;


/** Texture transform
 * translation and scale for mapping
 * texture points to vertices in square().
 **/
typedef struct textureTransform {
    float translate[2];
    float scale[2];
} textureTransform;


/** Square
 * a flat plane from x=0 to x=1, y=0 to y=1
 *
 * PARAMS:  p_front         material properties
 *          n_div           number of sub-divisions
 *          tex_transform   transform to apply to texture
 *
 **/
void square(const materialStruct* p_front, int n_div=1, textureTransform* tex_transform=0);


/** Cube
 * 9 outward-facing squares to form a cube.
 *
 * PARAMS:  p_front         material properties
 *
 **/
void cube(const materialStruct* p_front);


/** Cylinder
 * taken from COMP3811 tutorials
 * a cylinder of origin at the centre with z=0.
 * normals face inwards to reflect contained light source.
 *
 * PARAMS:  p_front         material properties
 *          N               number of faces
 *          n_div           number of height divisions
 *
 **/
void cylinder_inside(const materialStruct* p_front, int N=6, int n_div=1);


/** Sphere
 * GLUT sphere of origin at its centre.
 * normals face inwards to reflect contained light source.
 *
 * PARAMS:  p_front         material properties
 *
 **/
void sphere_inside(const materialStruct* p_front);

#endif
```

Figure 3: Definitions in `utils/Shapes.h`

```
4   void square(const materialStruct* p_front, int n_div, textureTransform* tex_transform) {
5       // set material properties
6       glMaterialfv(GL_FRONT, GL_AMBIENT,  p_front->ambient);
7       glMaterialfv(GL_FRONT, GL_DIFFUSE,  p_front->diffuse);
8       glMaterialfv(GL_FRONT, GL_SPECULAR, p_front->specular);
9       glMaterialf(GL_FRONT, GL_SHININESS, p_front->shininess);
10
11      float step_size = 1.0/n_div;
12
13      for (int i = 0; i < n_div; ++i)
14          for (int j = 0; j < n_div; ++j)
15          {
16              // vertex coordinates
17              float x0 = i * step_size;
18              float y0 = j * step_size;
19              float x1 = x0 + step_size;
20              float y1 = y0 + step_size;
21
22              // texture coordinates
23              float tx0 = x0;
24              float ty0 = y0;
25              float tx1 = x1;
26              float ty1 = y1;
27
28              // apply texture transforms (if present)
29              if (tex_transform) {
30                  tx0 = x0 + tex_transform->translate[0];
31                  ty0 = y0 + tex_transform->translate[1];
32                  tx1 = tx0 + step_size*tex_transform->scale[0];
33                  ty1 = ty0 + step_size*tex_transform->scale[1];
34              }
35
36              // create polygon
37              glBegin(GL_POLYGON);
38                  glTexCoord2f(tx0, ty0);
39                  glNormal3f(0,0,1);
40                  glVertex3f(x0,y0,0);
41
42                  glTexCoord2f(tx1, ty0);
43                  glNormal3f(0,0,1);
44                  glVertex3f(x1,y0,0);
45
46                  glTexCoord2f(tx1, ty1);
47                  glNormal3f(0,0,1);
48                  glVertex3f(x1,y1,0);
49
50                  glTexCoord2f(tx0, ty1);
51                  glNormal3f(0,0,1);
52                  glVertex3f(x0,y1,0);
53              glEnd();
54          }
55  }
```

Figure 4: `square()` implementation, `utils/Shapes.cpp`

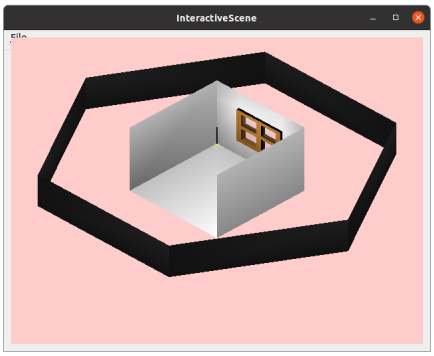


Figure 5: Orthographic view of complex scene (early implementation)

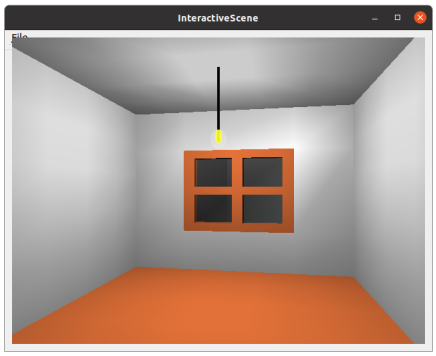


Figure 6: Perspective view of complex scene, no light attenuation

#### 2.2.1 Floor

A single instance of `square()` using the `woodMaterials` material properties. This was later textured with the `"wild_cherry_mysticBrown.png"` image.

#### 2.2.2 Walls

3 instances of `square()` for the top, left and right walls. The front wall is comprised of 8 instances of `square()` along the sides and diagonals of the window. This is done to ensure consistent interpolation of lighting without needing to implement new basic shapes or polygons. All of the walls use `whitePaintMaterials` and were later textured with `"Finishes.Painting.Paint.White.Flaking.jpg"`.

#### 2.2.3 Window

12 instances of `cube()`, scaled to cuboids, for the window "bones" and 9 instances of `cube()` for the window "joints". Again, segmentation was done to keep lighting consistent. These use the `woodMaterials` properties, and were later textured with the `"wild_cherry_mysticBrown.png"` image. The window panes are a single instance of `square()` using the `glassMaterials` properties, positioned inside of the window frame.

#### 2.2.4 Light bulb

A single instance of `cylinder_inside()` using the `blackPlasticMaterials` properties forms the cable and filament. The glass bulb and yellow light are instances of `sphere_inside()` using the semi-transparent `glassMaterials` and `warmLightMaterials` respectively.

#### 2.2.5 Background

A single instance of `cylinder_inside()` using the `backgroundMaterials` properties, giving it high ambient lighting. The texture of this was later implemented to be set via user interaction.

## 2.3 Materials and lighting

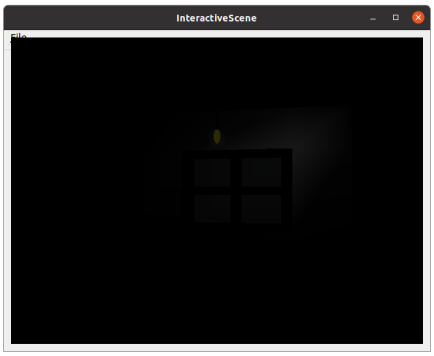


Figure 7: Perspective view of complex scene, specular light only

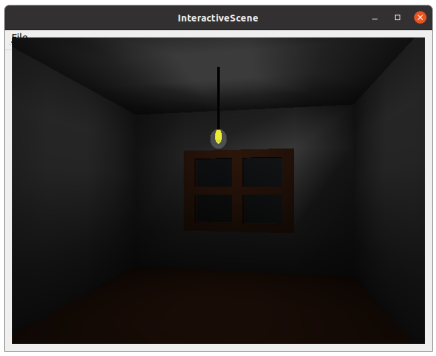


Figure 8: Perspective view of complex scene

- 6 materials of type `materialStruct` (Figure 3) are defined in Figure 9 representing all of the materials required for the scene. Most notably, the glass and light materials make use of the alpha channel for transparency, while background materials exhibit only ambient light properties.
- Alpha blending was used to draw the light bulb and window panes to add transparency and create a convincing glass-like appearance.
- 2 OpenGL lights are enabled in the scene, namely `GL_LIGHT0` to illuminate the room and `GL_LIGHT1` acting as the glow of the light bulb with high attenuation.
- Light attenuation was used in favour of spotlights to create a compelling visual atmosphere.

```
#ifndef __SC18J3J_MATERIALS__
#define __SC18J3J_MATERIALS__
#include "Shapes.h"

static materialStruct woodMaterials = {
    { 0.0, 0.0, 0.0, 1.0},
    { 0.6, 0.5, 0.4, 1.0},
    { 0.0, 0.0, 0.0, 1.0},
    20.0
};

static materialStruct whitePaintMaterials = {
    { 0.0, 0.0, 0.0, 1.0},
    { 0.5, 0.5, 0.5, 1.0},
    { 0.5, 0.5, 0.5, 1.0},
    50.0
};

static materialStruct blackPlasticMaterials = {
    { 0.0, 0.0, 0.0, 1.0},
    { 0.1, 0.1, 0.1, 1.0},
    { 0.6, 0.6, 0.6, 1.0},
    50.0
};

static materialStruct glassMaterials = {
    { 0.0, 0.0, 0.0, 1.0},
    { 0.6, 0.7, 0.8, 0.2},
    { 0.6, 0.7, 0.8, 0.2},
    100.0
};

static materialStruct warmLightMaterials = {
    { 1.0, 1.0, 0, 1.0},
    { 1.0, 1.0, 0, 0.9},
    { 1.0, 1.0, 0, 1.0},
    100.0
};

static materialStruct backgroundMaterials = {
    { 1, 1, 1, 1.0},
    { 0, 0, 0, 1.0},
    { 0, 0, 0, 1.0},
    100.0
};

#endif
```

Figure 9: Pre-defined materials in `utils/MaterialPredefs.h`

# 3 Band 3 (50%-60%)

The requirements of Band 3 are as follows:

1. The scene contains at least one element of user interaction

## 3.1 User interaction

| Name | Type | Values | Description |
|---|---|---|---|
| Light intensity | QSlider | range(0,20) | Magnitude of diffuse from light bulb (`GL_LIGHT0`) |
| Light bulb amplitude | QSlider | range(0,85) | Amplitude of light bulb sine curve |
| Light bulb period | QSlider | range(10,30) | Period of light bulb sine curve |
| Background texture | QComboBox | Dark woods/ Marc de Kamps/ Mercator projection | Select background image visible through window |
| Background speed | QSlider | range(-6,6) | Speed of background visible through window |
| Rocking chair speed | QSlider | range(0,30) | Speed of rocking chair |
| Head vibrate speed | QSlider | range(0,10) | Speed of head vibration |
| Orbit the room | QCheckBox | 0/1 | Character will orbit the room if set |
| Reset | QPushButton | None | Reset all widgets and geometry to default values |

Table 2: Qt user interface, tabulated

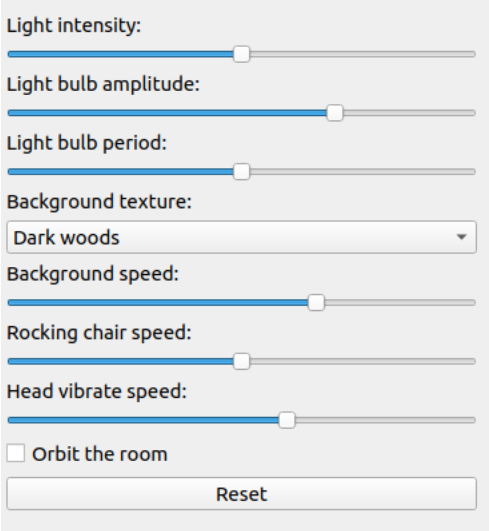


Figure 10: Qt user interface, actual

User interfaces provided through Qt (Table 2 and Figure 10) allow users to change visual aspects of the scene including light intensity and the texture of the background visible through the window. Various interfaces are also present that control animation described in 4.1, including adjustment of the light bulb's sine wave, speed of the background, rocking chair and head, as well as setting orbital movement. A `reset()`

function (Figure 13) sets all widgets and scene geometry to default values, and can be triggered via the "Reset" button.

Signals of each widget are connected to "setter" functions exposed as public slots in the `QGLWidget` such that changes in the user interface are almost immediately conveyed to the scene. The setup for this is shown in Figures 11 and 12 in the case of the light intensity slider, where each widget has a similar implementation.

As illustrated in Figures 11 and 12, a change in the value of the light intensity slider will emit a signal to the corresponding slot in the `QGLWidget`, which in turn directly sets the diffuse property of `GL_LIGHT0`.

```
27      // light intensity
28      light_intensity_label = new QLabel("Light intensity:");
29      light_intensity_slider = new QSlider(Qt::Horizontal);
30      light_intensity_slider->setRange(0,20);
31      connect(light_intensity_slider, &QSlider::valueChanged, scene_widget_, &SceneWidget::set_light_diffuse);
```

Figure 11: Setup of light intensity slider, `MainWindow.cpp`

```
317   void SceneWidget::set_light_diffuse(int value) {
318       GLfloat i = value/10.0;
319       GLfloat light_diffuse[] = {i, i, i, 1};
320       glLightfv(GL_LIGHT0, GL_DIFFUSE, light_diffuse);
321   }
```

Figure 12: Implementation of light intensity slot, `MainWindow.cpp`

```
111   void MainWindow::reset() {
112       light_intensity_slider->setValue(10);
113       light_bulb_amp_slider->setValue(60);
114       light_bulb_period_slider->setValue(20);
115       background_tex_combobox->setCurrentIndex(0);
116       background_speed_slider->setValue(2);
117       rocking_chair_speed_slider->setValue(15);
118       head_vibrate_speed_slider->setValue(6);
119       proof_of_orbit_checkbox->setChecked(0);
120       reset_geometry();
121   }
```

Figure 13: Implementation of `reset()`, `MainWindow.cpp`

# 4 Band 2 (60%-70%)

The requirements of Band 2 are as follows:

- an element of animation
- one convex object constructed from polygons
- texture mapping

## 4.1 Animation

Animation is created in the scene through a series of transformations over time. The `QGLWidget` rendering the scene is updated by a `QTimer` in the main window, aiming for a refresh rate of 100 frames per second. At each update, calculations are performed that adjust the transformations of objects to simulate movement in the scene (Figure 14). 2 variations of movement were implemented for this project as described below.

```
382     // set light bulb angle
383     light_bulb_dist = add_angle(light_bulb_dist, light_bulb_speed);
384     light_bulb_angle = sin(light_bulb_dist) * light_bulb_amplitude;
385     // background rotation
386     background_rotation += background_speed;
387     // set rocking chair angle
388     rocking_chair_dist = add_angle(rocking_chair_dist, rocking_chair_speed);
389     rocking_chair_angle = sin(rocking_chair_dist) * 20;
390     // set head angle
391     head_vibrate_dist = add_angle(head_vibrate_dist, head_vibrate_speed);
392     head_vibrate_angle = sin(head_vibrate_dist) * 10;
393     // set orbit angle
394     if (proof_of_orbit)
395         orbit_angle -= 1;
```

Figure 14: Distance and angle calculations performed on frame update, `SceneWidget.cpp`

### 4.1.1 Oscillation

Sine waves were used to animate oscillation in the OpenGL lights `GL_LIGHT0` and `GL_LIGHT1` and the light bulb object, creating the effect of a swinging light source. Sine waves were also used to generate oscillation in the character's rocking chair and head.

Oscillating objects are given a speed, where their angle of rotation at a given frame is a function of the distance travelled represented by a sine wave. All distances are updated at each frame update, and are normalised between the range of $-2\pi$ to $2\pi$ before they are passed to the sine function.

As an example, the plot of the light bulb angle as a function of distance using the default amplitude and period is shown in Figure 15.

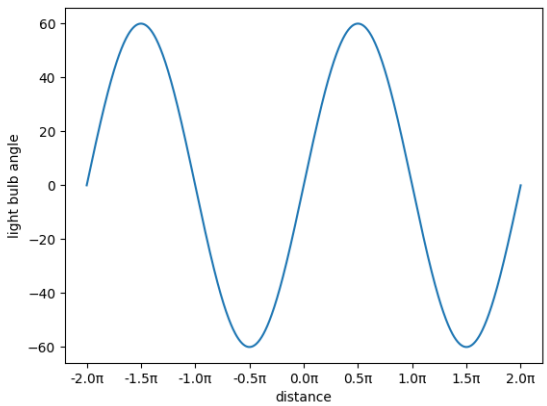


Figure 15: Light bulb angle as a function of distance

#### 4.1.2 Orbit

To animate an orbital motion around the room, we start at the centre point (0,0,0) and rotate by the orbit angle $\theta$ degrees in the z-axis. We then translate to (0,0.25,0) and draw the character. The orbit angle $\theta$ is decremented on each frame update if the QCheckBox labelled "Orbit the room" is set by the user. This creates circular movement in the clockwise direction around the scene. The background cylinder orbits the room in a similar manner, albeit without any translation.

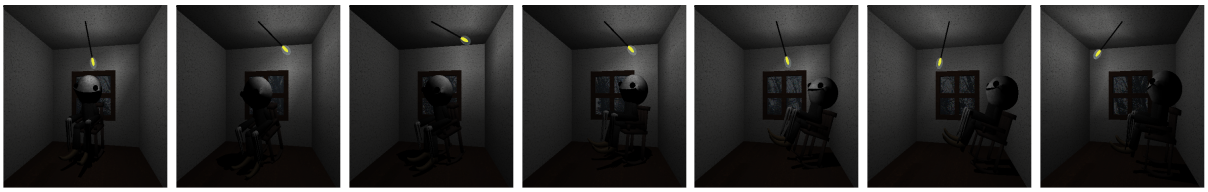

Figure 16: Animation in the scene (left to right)

### 4.2 Convex objects

A long-standing personal goal of mine is to learn how to create 3D art. This lined up nicely with the requirements of Band 2, and a convex object constructed from polygons was modelled using 3DS Max for the scene.

While most of the model is made up of basic convex shapes, complex geometry in the hands, face, shoes and rocking chair required the manipulation and creation of individual polygons which can be seen in the workspace view in Figure 18.

Figure 17: Render of our convex object

The head and body were exported separately in the Wavefront `.obj` format, and a `wavefrontObj` class was implemented in C++ to parse the files and draw their corresponding objects using OpenGL.

The `wavefrontObj` class implements 3 main functions, namely `draw()`, `load()` and `load_mtl()`. `draw()` handles the instantiation of the object in the scene, loading texture and material properties before emitting

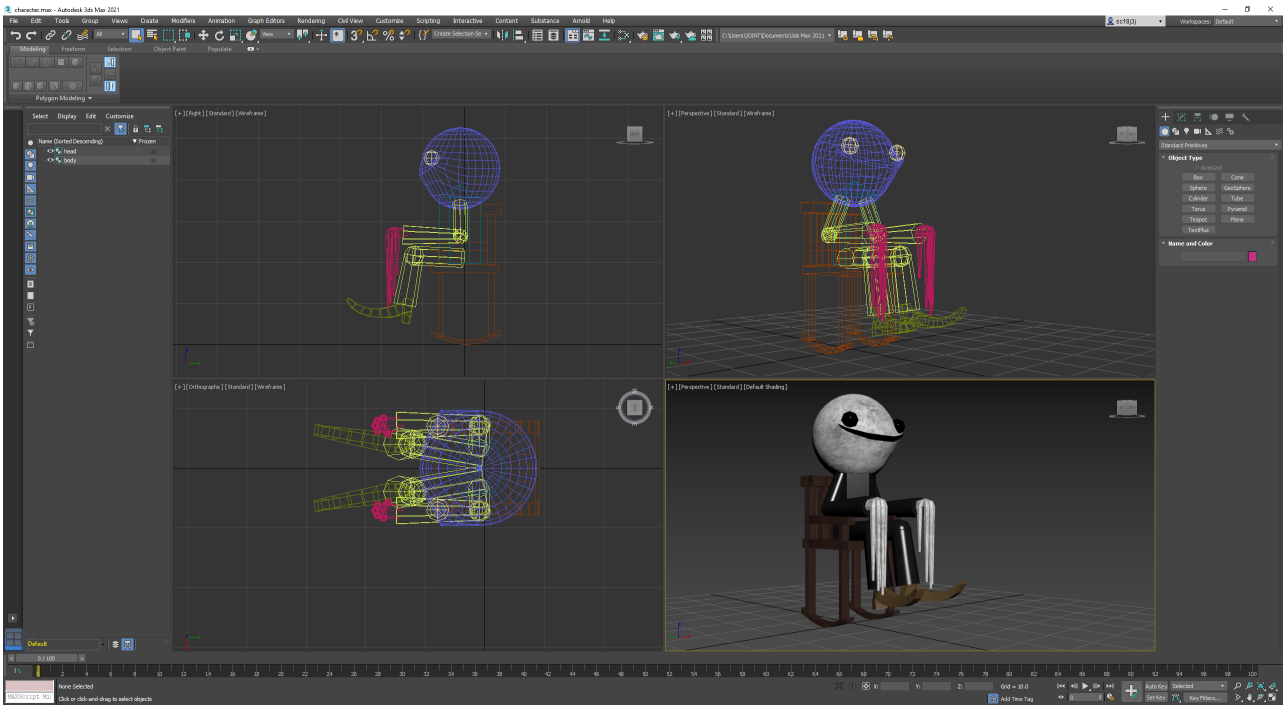

Figure 18: Workspace view of our convex object highlighting individual polygons

OpenGL vertex, normal and texture coordinates for each polygon (Figure 19). `load()` and `load_mtl()` parse input `.obj` and `.mtl` files and store the material, texture and coordinate information required by `draw()`.

An example of parsing a vertex is shown in Figure 20, where other information such as texture coordinates and material definitions are processed in a similar manner over a loop on every line in the input file.

On a side note, the definition of `cube()` in Figure 3 constitutes a convex object constructed from polygons, as it forms a solid cube from 9 instances of `square()`, which are themselves polygons created in OpenGL.

## 4.3 Texture mapping

The implementation of texture mapping in the scene is shown in Figures 3, 4 and 19. Images are loaded into the application using the `Image` class provided in the COMP3811 tutorials, and set in OpenGL with `glTexImage2D()`.

Instances of `square()` map the entire image to the polygon face of the square, subject to transformations specified by the `textureTransform` struct defined in Figure 3. The transform scales and offsets the image by a user-specified value to allow for a continuous image across multiple textured instances of `square()`.

Instances of `cylinder()` simply map the entire image to each polygon face of the cylinder.

Finally, texture mapping is performed on imported `.obj` models using texture information parsed from input files. Each sub-object in a `.obj` file can specify an image, stored in a `.mtl` file, and texture coordinates for each polygon vertex. As such, we simply load the corresponding image for each sub-object and emit the texture and vertex coordinates for each polygon to draw a textured 3D model.

```cpp
void WavefrontObj::draw() {
    // for each sub-object
    for (std::vector<wavefrontSubObj>::iterator sub_object = sub_objects->begin(); sub_object != sub_objects->end(); ++sub_object) {
        wavefrontMtl* mtl = sub_object->mtl;

        // if the sub-object's material has a path,
        // enable textures and load the corresponding image
        if (mtl->path.size()) {
            Image* texture = images->at(mtl->image_index);
            glEnable(GL_TEXTURE_2D);
            glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, texture->Width(), texture->Height(), 0, GL_RGB, GL_UNSIGNED_BYTE, texture->imageField());
        }

        // set material properties
        glMaterialfv(GL_FRONT, GL_AMBIENT,  mtl->ambient);
        glMaterialfv(GL_FRONT, GL_DIFFUSE,  mtl->diffuse);
        glMaterialfv(GL_FRONT, GL_SPECULAR, mtl->specular);
        glMaterialf(GL_FRONT, GL_SHININESS, mtl->shininess);

        // for each polygon
        for (uint face_index=0; face_index < sub_object->face_vertices.size(); ++face_index) {
            glBegin(GL_POLYGON);

            // set vertex, normal and texture coordinates
            for (uint point_index=0; point_index < sub_object->face_vertices.at(face_index).size(); ++point_index) {
                int v = sub_object->face_vertices.at(face_index).at(point_index);
                int t = sub_object->face_textures.at(face_index).at(point_index);
                int n = sub_object->face_normals.at(face_index).at(point_index);

                std::vector<float> vertex = vertices->at(v - 1);
                std::vector<float> texture = textures->at(t - 1);
                std::vector<float> normal = normals->at(n - 1);
                glTexCoord3f(texture.at(0), texture.at(1), texture.at(2));
                glNormal3f(normal.at(0), normal.at(1), normal.at(2));
                glVertex3f(vertex.at(0), vertex.at(1), vertex.at(2));
            }

            glEnd();
        }

        glDisable(GL_TEXTURE_2D);
    }
}
```

Figure 19: Implementation of `wavefrontObj::draw()`, `utils/WavefrontObj.cpp`

```cpp
// parse vertex
if (!strcmp(token, "v")) {
    std::vector<float> vertex;
    float x,y,z;

    if (sscanf(line.c_str(), "%s %f %f %f", token, &x, &y, &z) <= 1)
        continue;

    // store values
    vertex.push_back(x);
    vertex.push_back(y);
    vertex.push_back(z);
    vertices->push_back(vertex);
}
```

Figure 20: Implementation of parsing a vertices in a `.obj` file, `utils/WavefrontObj.cpp`

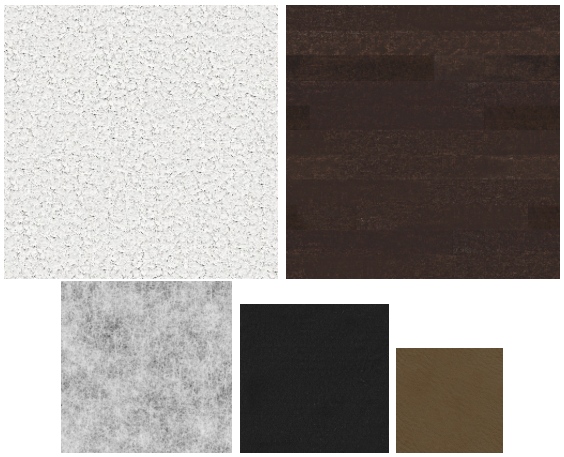

Figure 21: Autodesk textures

Figure 22: Dark woods texture, photographed in Chapel Allerton

Figure 23: Background texture visible through window

A number of images were introduced to texture different elements of the scene. The images shown in Figure 21 were taken from the default material library included with 3DS Max.

The image of Dr Marc de Kamps and the Mercator Projection were used as background textures visible through the window of the scene (Figure 23), with an additional photograph taken in Leeds (Figure 22) with the intention of adding to the visual atmosphere.

The choice of background image in the scene can be made via the user interface detailed in section 3, with the code that loads the selected image shown in Figure 24.

```
251 void SceneWidget::background() {
252     // load the selected background image
253     glEnable(GL_TEXTURE_2D);
254     glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, bg_textures[bg_index]->Width(), bg_textures[bg_index]->Height(), 0, GL_RGB, GL_UNSIGNED_BYTE, bg_textures[bg_index]->imageField());
255
256     // draw a large textured cylinder
257     glPushMatrix();
258     glRotatef(background_rotation, 0, 0, 1);
259     glTranslatef(0,0,-0.2);
260     glScalef(1.5,1.5,1.5);
261     cylinder_inside(&backgroundMaterials);
262     glPopMatrix();
263
264     glDisable(GL_TEXTURE_2D);
265 }
```

Figure 24: Loading the background texture image in `SceneWidget.cpp`

# 5 Band 1 (70%-100%)

The requirements of Band 1 are as follows:

- The scene contains an object that requires hierarchical modelling and displays motion in some of its parts.
- Various elements of user interaction are used.

## 5.1 Hierarchical modelling

As explained in section 4.2 our character was exported in two parts, namely its head and body. This allows us to perform hierarchical modelling where the head and the body can be drawn as children of the character instance, with their pose dependent on that of the parent.

This is shown clearly through animation, where the character can be set to orbit around the room while its entire body rocks with its head rotating in place. All of this is seen to occur without changing the relative position of the head to the character origin as a result of hierarchical modelling.

```
267  void SceneWidget::character() {
268      // rotate along x-axis for rocking chair
269      glRotatef(rocking_chair_angle,1,0,0);
270      body.draw();
271
272      // head is at z=2.15 units
273      glTranslatef(0,0,2.15);
274
275      // rotate along y and z axes for head vibration
276      glRotatef(head_vibrate_angle,0,1,0);
277      glRotatef(head_vibrate_angle,0,0,1);
278      head.draw();
279  }
```

Figure 25: Hierarchical implementation of the character object in `SceneWidget.cpp`

The implementations of `cube()` and `house()` also make use of hierarchical modelling, where the former creates a convex object out of multiple squares, and the latter creates a complex scene using a variety of basic shapes, all relative to the origin of the instance.

## 5.2 User interaction

Users are able to interact with the hierarchical model by setting the head vibration speed, rocking chair speed and its orbit around the room using the interfaces outlined in section 3. Sections 4.1 and 5.1 detail the process of animating these movements in the character.

## 5.3 Shadow implementation

Without the use of a shader, we model shadows in the scene by applying transformations to the model-view matrix that shear and flatten objects drawn onto the scene.

Setting the z coordinate to 0 in the model-view matrix flattens the z-axis, allowing us to draw on a 2D plane where z=0.

A positive z-shear in the x-axis draws the shadow to the left of the character, while a negative z-shear in the x-axis draws it to the right. Similarly, a positive z-shear in the y-axis draws the shadow behind the character, while a negative z-shear in the y-axis draws it in front.

We can thus model the direction of the shadow given the state of the scene, using sine and cosine waves to estimate the length of the z-shear in both axes from the character's position along its orbit in relation to the light source, and sine waves to approximate z-shear in the y-axis from rotation of the rocking chair.

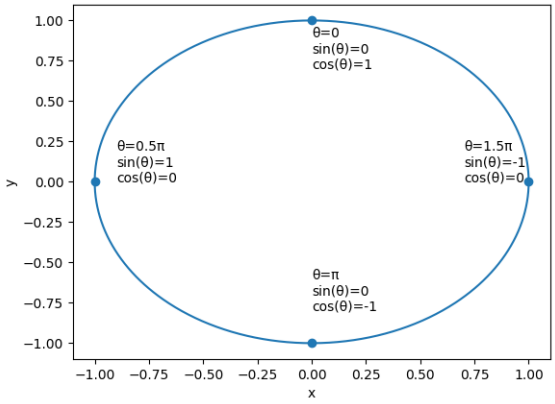


Figure 26: Plot of character orbit labelled with orbit angle $\theta$ (normalised) in radians.

The computed z-shears and flattening effect mentioned above are combined into a single matrix to form our shadow transformation matrix. We apply this transformation to the model-view matrix, temporarily disable depth testing and lighting and set the colour to (0,0,0,1) before drawing our character model. This has the effect of generating a flat dark shadow of the character on the floor where z=0.

The code for calculating the shadow transform can be found in Figure 27, while Figure 28 features some interesting shadows generated from this effect.

```
281  void SceneWidget::shadow() {
282      glPushMatrix();
283
284      // stop materials and z-fighting
285      glDisable(GL_LIGHTING);
286      glDisable(GL_DEPTH_TEST);
287
288      // calculate current angle of character
289      float character_angle_rad = (orbit_angle + character_tilt) * M_PI/180.0;
290
291      // shear when the light is swinging orthogonal to the character.
292      float shear_light = cos(character_angle_rad) * -light_bulb_angle/100.0;
293
294      // shear from chair rocking and when the light
295      // is swinging parallel to the character.
296      float shear_chair = 0.5 * (-sin(rocking_chair_dist) + 1) + sin(character_angle_rad) * light_bulb_angle/100.0;
297
298      /** Shadow transformation
299       * mathematical model to create character's shadow.
300       * performs z-shear in x,y axes and flattens the z-axis.
301       *
302       */
303      GLfloat shadow_transform[16] = {1,0,0,0,
304                                      0,1,0,0,
305                                      shear_light,shear_chair,0,0,
306                                      0,0,0,1};
307
308      // apply transform and draw the shadow in black (very hitchcock)
309      glMultMatrixf(shadow_transform);
310      glColor3f(0,0,0);
311      character();
312      glEnable(GL_LIGHTING);
313      glEnable(GL_DEPTH_TEST);
314      glPopMatrix();
315  }
```

Figure 27: implementation of shadows in `SceneWidget.cpp`

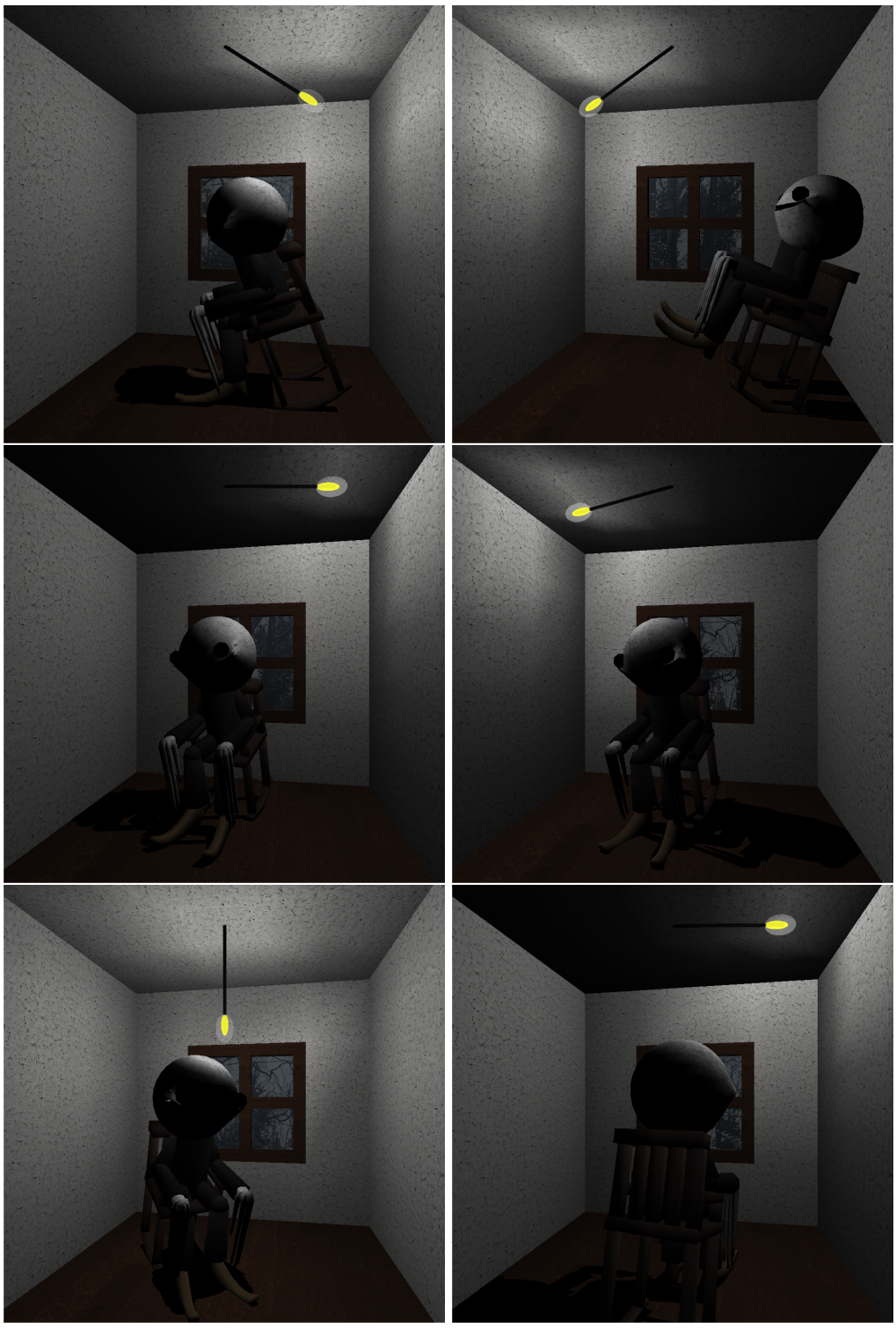

Figure 28: Shadows generated from the shadow transform

# 6 Conclusion

Overall I feel that I have been able to meet the technical requirements of the project, having implemented various techniques in OpenGL to draw, texture and illuminate objects in a 3D animated scene, as well as the accompanying Qt user interface to interact with it. I was also able to learn how to create 3D models from convex shapes and polygons using 3DS Max.

The general feedback from users on viewing the scene hint at discomfort, surprise and creepiness. This is in my opinion a success in creating a horrific atmosphere set out in the aims of the project, as the scene has been able to illicit similar emotions to the works I have drawn inspiration from.

## 6.1 Source code instructions

To run the application from source, enter the following commands from the project directory in a terminal:

```
cd src
qmake
make
./InteractiveScene
```

## 6.2 Video demonstrations

Software was tested using feng-linux/gpu however unfortunately subject to lower frame rates through the service. Video demonstrations of the scene running at full-speed are available here:

- https://youtu.be/I-vPSETKljw
- https://youtu.be/YgDw-1BfJhA

## 6.3 Closing scene

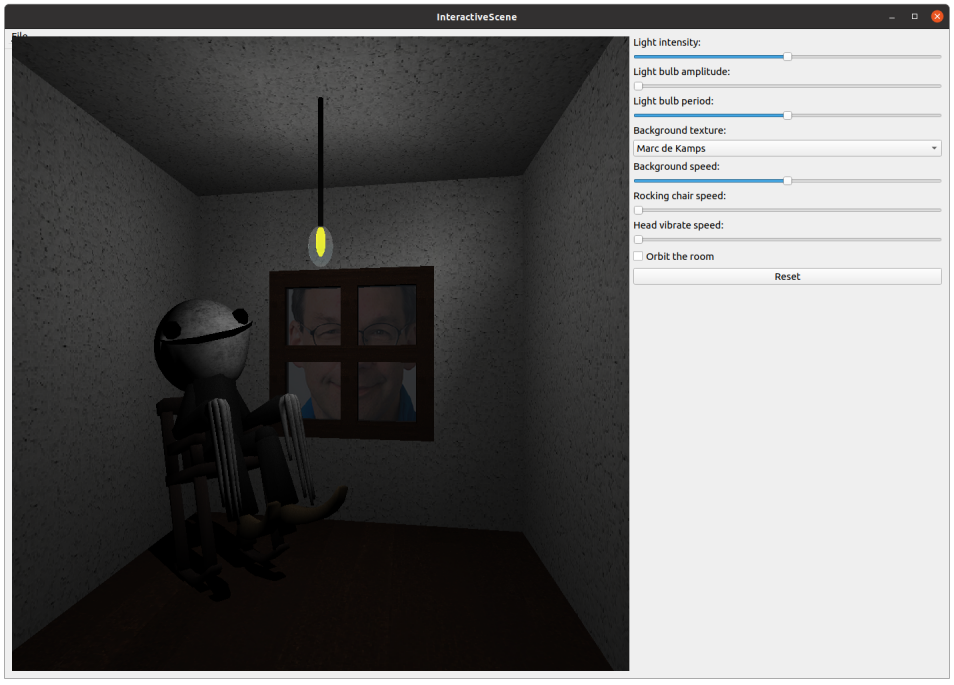


Figure 29: Thank you for reading this report!