110612117 張仲瑜 HW2 report

整體流程簡介：

1. 建立並編譯 shaders，再將兩者連接為著色器
2. 載入模型紋理並綁定
3. 設定 VAO 與 VBO 供頂點綁定
4. 讀取 uniform 變數位置
5. 開始根據初始化變數製圖並更新 uniform 參數

(圖零). main 參數

```
30      // You can use these parameters
31      float swingAngle = 0;
32      float swingPos = 0;
33      //float swingAngleDir = -1;
34      //float swingPosDir = 1;
35      float tmp_swingAngle = 0;
36      float tmp_swingPos = 0;
37      float squeezeFactor = 0;
38      float boardSqueezeFactor = 0;
39      int penguinNum = 1;
40      int penguinSide = 1;
41      int step = 0;
42      bool squeezing = false;
43      bool useGrayscale = false;
44      bool whitePenguin = false;
45      bool boardSqueezing = false;
```

依序為轉動角度與累積、位移與累積、擠壓參數、

企鵝數量、企鵝位移方向、企鵝編號暫存、是否擠壓企鵝

、是否擠壓衝浪板、是否灰階、是否白化

```
89      unsigned int vertexShader, fragmentShader, shaderProgram;
90      vertexShader = createShader("vertexShader.vert", "vert");
91      fragmentShader = createShader("fragmentShader.frag", "frag");
92      shaderProgram = createProgram(vertexShader, fragmentShader);
```

(圖一). 建立後面要用到的著色程式

透過 createShader function 建立 vertex shader

與 fragmentShader, 再透過 createProgram function

將他們連成一個著色程式

```
324 unsigned int createShader(const string& filename, const string& type)
325 {
326     GLuint shader = glCreateShader(type == "vert" ? GL_VERTEX_SHADER : GL_FRAGMENT_SHADER);
327
328     ifstream shaderFile(filename);
329     stringstream shaderStream;
330     shaderStream << shaderFile.rdbuf();
331     string shaderCode = shaderStream.str();
332     const char* shaderSource = shaderCode.c_str();
333
334     glShaderSource(shader, 1, &shaderSource, NULL);
335     glCompileShader(shader);
336
337     return shader;
```

(圖二). CreateShader function

Line 326

透過判斷傳進來的文字參數，來決定要建立的 shader 類型.

Line 328~332

讀取傳入檔案內的內容轉換成字串，並取得他的指標

Line 334~336

透過 glShaderSource(建立位置，字串行數，字串內容，字串符結尾）建立 shader，編譯後再傳回

```
345 unsigned int createProgram(unsigned int vertexShader, unsigned int fragmentShader)
346 {
347     GLuint program = glCreateProgram();
348
349     glAttachShader(program, vertexShader);
350     glAttachShader(program, fragmentShader);
351
352     glLinkProgram(program);
353
354     int isSuccess;
355     char infoLog[512];
356     glGetProgramiv(program, GL_LINK_STATUS, &isSuccess);
357     if (!isSuccess) {
358         glGetProgramInfoLog(program, 512, NULL, infoLog);
359         cout << "Shader program linking failed: " << infoLog << endl;
360     }
361
362     glDeleteShader(vertexShader);
363     glDeleteShader(fragmentShader);
364     return program;
```

(圖三). CreateShader function

Line 349~352

建立一個 program，將兩個 shader 加入後連結起來

Line 354~360

確認是否連結成功

連結失敗就輸出失敗紀錄

Line 362~364

刪除已連結完的 shader，回傳著色器程式

```
unsigned int penguinTexture, boardTexture;
penguinTexture = loadTexture("obj/penguin_diffuse.jpg");
boardTexture = loadTexture("obj/surfboard_diffuse.jpg");
```

```
371 unsigned int loadTexture(const char* tFileName) {
372     unsigned int texture;
373     glEnable(GL_TEXTURE_2D);
374     glActiveTexture(GL_TEXTURE0);
375     glGenTextures(1, &texture);
376     glBindTexture(GL_TEXTURE_2D, texture);
377
378     glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
379     glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
380
381     int width, height, nrChannels;
382     stbi_set_flip_vertically_on_load(true);
383     unsigned char* data = stbi_load(tFileName, &width, &height, &nrChannels, 0);
384     if (data) {
385         //glBindTexture(GL_TEXTURE_2D, texture);
386         glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0, GL_RGB, GL_UNSIGNED_BYTE, data);
387     }
388     else {
389         cout << "Loading texture failed" << endl;
390     }
391     stbi_image_free(data);
392     return texture;
```

(圖四、五). 宣告企鵝與衝浪板的紋理，透過讀取 function 設定指定的圖片紋理

Line 373~376

啟用 2D 紋理、指定啟用的紋理編號、產生 1 個紋理並將他綁到指定的 2D 紋理

Line 378~379

設置紋理縮小時的過濾方式為線性過濾

Line 381~383

宣告變數來保存圖像的寬度、高度和通道數，

使用 stb 圖像庫讀取圖片文件然後存在 data 變數中

Line 384~391

確認是否讀取成功，讀取失敗就輸出失敗紀錄

成功就釋放記憶體並回傳紋理變數

```
108   unsigned int penguinVAO, boardVAO;
109   penguinVAO = modelVAO(penguinModel);
110   boardVAO = modelVAO(boardModel);
```

```
400 unsigned int modelVAO(Object& model)
401 {
402     unsigned int VAO, VBO[3];
403
404     glGenVertexArrays(1, &VAO);
405     glBindVertexArray(VAO);
406     glGenBuffers(3, VBO);
407
408     glBindBuffer(GL_ARRAY_BUFFER, VBO[0]);
409     glBufferData(GL_ARRAY_BUFFER, sizeof(GL_FLOAT) * (model.positions.size()), &(model.positions[0]), GL_STATIC_DRAW);
410     glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, sizeof(GL_FLOAT) * 3, 0);
411     glEnableVertexAttribArray(0);
412     glBindBuffer(GL_ARRAY_BUFFER, 0);
413
414
415     glBindBuffer(GL_ARRAY_BUFFER, VBO[1]);
416     glBufferData(GL_ARRAY_BUFFER, sizeof(GL_FLOAT) * (model.normals.size()), &(model.normals[0]), GL_STATIC_DRAW);
417     glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, sizeof(GL_FLOAT) * 3, 0);
418     glEnableVertexAttribArray(1);
419     glBindBuffer(GL_ARRAY_BUFFER, 0);
420
421     glBindBuffer(GL_ARRAY_BUFFER, VBO[2]);
422     glBufferData(GL_ARRAY_BUFFER, sizeof(GL_FLOAT) * (model.texcoords.size()), &(model.texcoords[0]), GL_STATIC_DRAW);
423     glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, sizeof(GL_FLOAT) * 2, 0);
424     glEnableVertexAttribArray(2);
425     glBindBuffer(GL_ARRAY_BUFFER, 0);
426
427     glBindVertexArray(0);
428     return VAO;
```

(圖五、六). ModelVAO 內容

entShader.frag | vertexShader.vert | Main.cpp*

```
1   #version 330 core
2
3   layout (location = 0) in vec3 aPos;
4   layout (location = 1) in vec3 aNormal;
5   layout (location = 2) in vec2 aTexCoord;
```

(圖七) shader 內對應屬性索引值

宣告企鵝與衝浪板的 VAO

再透過 model VAO 指定 VAO 與 VBO 裡面的格式

其中一個 VAO 對應到三個 VBO 分別存儲頂點的位置、法線和紋理座標數據

glBindBuffer

綁定 GL_ARRAY_BUFFER 到當前的 VBO

glBufferData(目前綁定的 buffer 對象,要複製的字節大小, 要複製的位置來源, 渲染一次並多次使用)

glVertexAttribPointer(VertexShader 裡該屬性的索引值, 屬性大小(三個頂點/texcoord 則為 U、V 二維), 屬性型別, 要不要正規化, 兩個屬性間的步長, 起始的偏移)

Line 425~428

啟用頂點的屬性陣列

一切都完成後解除綁定緩衝區並回傳 VAO

```
121     /* TODO#4: Data connection - Retrieve uniform variable locations
122     *    1. Retrieve locations for model, view, and projection matrices.
123     *    2. Obtain locations for squeezeFactor, grayscale, and other parameters.
124     * Hint:
125     *    glGetUniformLocation
126     */
127     unsigned int modelMatrixLocation = glGetUniformLocation(shaderProgram, "M");
128     unsigned int viewMatrixLocation = glGetUniformLocation(shaderProgram, "V");
129     unsigned int projectionMatrixLocation = glGetUniformLocation(shaderProgram, "P");
130     unsigned int squeezeFactorLocation = glGetUniformLocation(shaderProgram, "squeezeFactor");
131     unsigned int boardSqueezeFactorLocation = glGetUniformLocation(shaderProgram, "boardSqueezeFactor");
132     unsigned int ourTextureLocation = glGetUniformLocation(shaderProgram, "ourTexture");
133     unsigned int useGrayscaleLocation = glGetUniformLocation(shaderProgram, "useGrayscale");
134     unsigned int whitePenguinLocation = glGetUniformLocation(shaderProgram, "whitePenguin");
```

(圖八) 讀取 shader 裡的各項 uniform 變數的位置

設定變數將他們存起來。

接下來介紹 shader 中的變數與 shader 作用

```
3     layout (location = 0) in vec3 aPos;
4     layout (location = 1) in vec3 aNormal;
5     layout (location = 2) in vec2 aTexCoord;
6
7     uniform mat4 M;
8     uniform mat4 V;
9     uniform mat4 P;
10
11    uniform float squeezeFactor;
12    uniform float boardSqueezeFactor;
13
14    out vec2 texCoord;
15    out vec3 normal;
16
17    vec4 worldPos;
```

(圖九) Vertex shader 的輸入、輸出與統一變數

輸入：位置、法向量、紋理座標

輸出(至 fragment shader)：紋理座標、法向量，將在後續內差後送至 fragment shader

統一變數：模型矩陣、視角矩陣、投影矩陣、企鵝

衝浪板擠壓程度

```
28          //step1
29          vec3 squeezedPos = aPos;
30          squeezedPos.x += squeezedPos.z * sin(boardSqueezeFactor) / 2.0;
31          squeezedPos.z += squeezedPos.y * sin(boardSqueezeFactor) / 2.0;
32          squeezedPos.y += squeezedPos.z * sin(squeezeFactor) / 2.0;
33          squeezedPos.z += squeezedPos.y * sin(squeezeFactor) / 2.0;
34          //step2
35          worldPos = M * vec4(squeezedPos, 1.0);
36          //step 3
37          gl_Position = P * V * worldPos;
38          //step 4
39          texCoord = aTexCoord;
40          normal = mat3(transpose(inverse(M)))* aNormal;
41      }
```

(圖十) Vertex shader 執行的事情

Line 29~33

將頂點座標根據擠壓公式轉換後存起來

Line 35~37

將擠壓後的頂點座標乘上 MPV 轉換成最後的座標

Line 39~40

將紋理座標與轉換後的法向量存給要輸出給 fragment shader 的變數

```
3       in vec2 texCoord;
4       in vec3 normal;
5
6       uniform sampler2D ourTexture;
7       uniform bool useGrayscale;
8       uniform bool whitePenguin;
9       out vec4 FragColor;
```

(圖十一) fragment shader 的輸入、輸出與統一變數

輸入：vertex shader 的輸出

輸出：每個像素最終顏色值

統一變數：2D 紋理、是否灰階/白化的 boolean 值

```
21          vec4 color = texture2D(ourTexture, texCoord);
22          //step 2
23          if(useGrayscale && !whitePenguin){
24              //step 2-1
25              float grayScale = dot(color.rgb, vec3(0.299, 0.587, 0.114));
26              //step 2-2
27              FragColor = vec4(vec3(grayScale), color.a);
28          }
29          else if(whitePenguin){
30              float white = dot(color.rgb, vec3(1.0, 1.0, 1.0));
31              FragColor = vec4(vec3(white), color.a);
32          }
33          else{
34              FragColor = color;
35          }
36      }
```

(圖十二) fragment shader 執行的事情

Line 21

根據紋理中根據座標取得顏色值

Line 23~36

如果有啟用灰階/白化，則將 RGB 三者透過灰階/白化轉換公式與透明度形成最後的像素顏色值

否則直接將原本的顏色值輸出

```
154         glUseProgram(shaderProgram);
155         glm::mat4 surfBoard = glm::mat4(1.0f);
156         surfBoard = glm::translate(surfBoard, glm::vec3(0.0f, -0.5f, 0.0f));
157         surfBoard = glm::rotate(surfBoard, glm::radians(swingAngle), glm::vec3(0.0f, 1.0f, 0.0f));
158         surfBoard = glm::translate(surfBoard, glm::vec3(0.0f, 0.0f, swingPos));
159         surfBoard = glm::rotate(surfBoard, glm::radians(-90.0f), glm::vec3(1.0f, 0.0f, 0.0f));
160         surfBoard = glm::scale(surfBoard, glm::vec3(0.03f, 0.03f, 0.03f));
161         glUniform1f(useGrayscaleLocation, useGrayscale);
162         glUniform1f(whitePenguinLocation, whitePenguin);
163         glUniform1f(squeezeFactorLocation, 0);
164         glUniform1f(boardSqueezeFactorLocation, glm::radians(boardSqueezeFactor));
165         glUniformMatrix4fv(modelMatrixLocation, 1, GL_FALSE, value_ptr(surfBoard));
166         glUniformMatrix4fv(viewMatrixLocation, 1, GL_FALSE, value_ptr(view));
167         glUniformMatrix4fv(projectionMatrixLocation, 1, GL_FALSE, value_ptr(perspective));
168
169         glActiveTexture(GL_TEXTURE1);
170         glBindTexture(GL_TEXTURE_2D, boardTexture);
171         glUniform1i(ourTextureLocation, 1);
172
173         glBindVertexArray(boardVAO);
174         glDrawArrays(GL_TRIANGLES, 0, boardModel.positions.size());
175         glBindVertexArray(0);
176
177         glUseProgram(0);
```

(圖十三) 衝浪板的產圖過程

Line 154

啟用前面做好的著色 program

Line 156~160

根據要求將衝浪板擺放、縮放與旋轉

Line 161~164

設定衝浪板行為對應的 uniform 變數

因應創意項將 board 設定獨立的擠壓變數

Line 165~167

將 surfBoard、view、perspective 三個矩陣設定給 vertex shader 裡對應的 uniform M、V、P 變數

參數(uniform 變數位置, 矩陣個數, 是否將矩陣轉置, 要轉

換格式後傳入的矩陣指標)

Line 169~172

指定啟用的紋理單元並且將衝浪板紋理綁定上去

並將 uniform texture 變數設定為我們剛剛啟用的單元索引

Line 173~177

綁定 VAO 後開始繪製，繪製模式為三角形繪製

從第 0 個點開始繪製，與繪製的頂點數量

完成後解綁 VAO、解綁著色程式

```
189         glUseProgram(shaderProgram);
190         penguinSide = 1;
191         step = 0;
192         for(int i = 0; i < penguinNum; i++){
193             float size;
194             if (i % 2 == 1) {
195                 step++;
196             }
197             float Offset = 1.5f * step * penguinSide + 0.0f;
198             penguinSide = -penguinSide;
199             size = 0.025f / (step + 1);
200             glm::mat4 penguin = glm::mat4(1.0f);
201             penguin = glm::translate(penguin, glm::vec3(Offset, 0.0f, 0.0f));
202             penguin = glm::rotate(penguin, glm::radians(swingAngle), glm::vec3(0.0f, 1.0f, 0.0f));
203             penguin = glm::translate(penguin, glm::vec3(0.0f, 0.0f, swingPos));
204             penguin = glm::rotate(penguin, glm::radians(-90.0f), glm::vec3(1.0f, 0.0f, 0.0f));
205             penguin = glm::scale(penguin, glm::vec3(size, size, size));
206             glUniform1f(whitePenguinLocation, whitePenguin);
207             glUniform1f(useGrayscaleLocation, useGrayscale);
208             glUniform1f(squeezeFactorLocation, glm::radians(squeezeFactor));
209             glUniform1f(boardSqueezeFactorLocation, 0);
210             glUniformMatrix4fv(modelMatrixLocation, 1, GL_FALSE, value_ptr(penguin));
211             glUniformMatrix4fv(viewMatrixLocation, 1, GL_FALSE, value_ptr(view));
212             glUniformMatrix4fv(projectionMatrixLocation, 1, GL_FALSE, value_ptr(perspective));
213 
214             glActiveTexture(GL_TEXTURE2);
215             glBindTexture(GL_TEXTURE_2D, penguinTexture);
216             glUniform1i(ourTextureLocation, 2);
217 
218             glBindVertexArray(penguinVAO);
219             glDrawArrays(GL_TRIANGLES, 0, penguinModel.positions.size());
220         }
221         glBindVertexArray(0);
222         glUseProgram(0);
```

(圖十五) 企鵝的產圖過程

整體與衝浪板大致相同

這裡我著重在不同處與創意項介紹

企鵝需要考慮是否採用擠壓、白化、灰階等

因此在 Line 207

將 squeezefactor 的 radians 傳給對應的 uniform 變數

衝浪板的 squeezefactor 則設為零

在創意項我設定了按下 m 鍵時

會在衝浪板的左右兩側產生兩隻小企鵝

因此透過迴圈重複產生企鵝

其中當 penguinNum = 1 時

位移量與縮放皆等於原本設定項(step num 皆是 0)

而企鵝需以一左一右產生

所以 penguin side 每次繪圖會正負相間轉換

大小則會對稱縮小

所以 step 每繪圖兩次增加一次

礙於效能問題

我將總企鵝量上限設定為 3 隻

超過三隻即會回到原狀

```
231         tmp_swingAngle += 20.0f * static_cast<float>(dt);
232         tmp_swingPos += 1.0f * static_cast<float>(dt);
233         float SwingPeriodAng = 2 * (20.0f - (- 20.0f));
234         float aQuarterAng = SwingPeriodAng / 4;
235         float threeQuarterAng = 3 * aQuarterAng;
236         float HalfSwingPeriodAng = 2 * aQuarterAng;
237
238         float SwingPeriodPos = 2 * (2.0f - (0.0f));
239         float aQuarterPos = SwingPeriodPos / 4;
240         float threeQuarterPos = 3 * aQuarterPos;
241         float HalfSwingPeriodPos = 2 * aQuarterPos;
```

```
243     if (squeezing) {
244         squeezeFactor += 90.0f * static_cast<float>(dt);
245     }
246
247     if (tmp_swingAngle >= SwingPeriodAng) {
248         tmp_swingAngle -= SwingPeriodAng;
249     }
250
251     if (tmp_swingAngle >= threeQuarterAng) {
252         swingAngle = -SwingPeriodAng + tmp_swingAngle;          //-(80 - 60~80)
253     }
254     else if (tmp_swingAngle >= aQuarterAng) {
255         swingAngle = HalfSwingPeriodAng - tmp_swingAngle;   //40-(20~60)
256     }
257     else {
258         swingAngle = tmp_swingAngle;
259     }
```

```
262     if (tmp_swingPos >= SwingPeriodPos) {
263         tmp_swingPos -= SwingPeriodPos;
264     }
265
266     if (tmp_swingPos >= HalfSwingPeriodPos) {       //2~4
267         swingPos = (SwingPeriodPos - tmp_swingPos) ;    //(4 - 2~4)
268     }
269     else {
270         swingPos = tmp_swingPos;
271     }
```

(圖十六、十七、十八)

衝浪板與企鵝的擺動、位移與壓縮量的變化

Line 231 ~ 232、Line 243~245

根據要求每秒增加值，擺動與位移因為需要將區間轉換

設定同名分身操作後再存回繪製時用的變數

Line 233~241

根據區間轉換週期設定各 1,2,3,4,/4 週期的長度

Line 247~271

如果週期，則週期重啟

推導後將每週期段對應的處理更新到暫存參數上後

再更新繪製時用的變數

以角度為例

暫存 0~20 度的時候 轉 0~20 度

暫存 20~40 度的時候轉 20~0 度

暫存 40~60 度的時後轉 0~–20 度

暫存 60~80 度的時候轉–20~0 度

以此循環

位移概念也相同，只是範圍不同

```
295  void keyCallback(GLFWwindow* window, int key, int scancode, int action, int mods)
296  {
297      if (key == GLFW_KEY_ESCAPE && action == GLFW_PRESS) {   //GLFW means graphic library framework
298          glfwSetWindowShouldClose(window, true);
299      }
300      else if (key == GLFW_KEY_S && action == GLFW_PRESS) {
301          squeezing = !squeezing;
302          cout << "KEY S PRESSED" << endl;
303      }
304      else if (key == GLFW_KEY_G && action == GLFW_PRESS) {
305          useGrayscale = !useGrayscale;
306          cout << "KEY G PRESSED" << endl;
307      }
308      else if (key == GLFW_KEY_M && action == GLFW_PRESS) {
309          if (penguinNum < 3) {
310              penguinNum++;
311          }
312          else {
313              penguinNum = 1;
314          }
315          cout << "KEY M PRESSED" << endl;
316      }
317      else if (key == GLFW_KEY_W && action == GLFW_PRESS) {
318          whitePenguin = !whitePenguin;
319          cout << "KEY W PRESSED" << endl;
320      }
321      else if (key == GLFW_KEY_B && action == GLFW_PRESS) {
322          boardSqueezing = !boardSqueezing;
323          cout << "KEY B PRESSED" << endl;
324      }
```

(圖十九) 按鍵觸發事件

按下 S 時將企鵝擠壓開關相反

按下 B 將板子擠壓相反

按下 W 時將白化開關相反

按下 G 時將灰階開關相反

按下 M 時增加企鵝數量

若企鵝數量達到 3 就回到 1

過程遇到的問題/心得

在紋理讀取中，我一開始將綁定放在讀取成功後才執行，但這樣衝浪板一直繪製不出來，推測是會使用到預設的紋理單元然後被覆蓋掉了，除錯了很久才發現是這邊導致的問題；此外，在旋轉與的參數映射上也試錯很久，直覺上反方向轉就應該是負數，然而在 1/4~2/4 週期向後轉是正數減少而不是負數，這解決了我的企鵝瞬間移動的問題。這次作業讓本來比較抽象的概念，透過實作整合具現化了一些，儘管過程中處處碰壁，但在做出成品的瞬間成就感十足，是難能可貴的經驗，也十分期待下次的作業。