四資工二甲　B11132021　蔣博元

電腦圖學 Project 03 – Checkpoint 01

目次

TOP

API 文件

詳見 doc/index.chm

Draw.h

Draw.h 的函數都宣告在 namespace Draw 內。

- Draw::Param_Equation – typedef，用來表示參數式的 function object
- Draw::make_line – 建立直線參數式
- Draw::make_cubic_b_spline – 建立 Cubic B-Spline 的參數式
- Draw::make_cardinal – 建立 Cardinal Spline 的參數式
- Draw::set_equation – 設定兩 control point 間的參數式
- Draw::draw_track – 畫軌道
- Draw::draw_train – 畫火車

Enum.h

包含了一些全域變數和 enum。

- class SplineType – 表示繪製時使用的 Spline 種類
- enum LightType – 表示光源的種類
- namespace GLOBAL – 內含全域變數

Track.h

我有在 CTrack 內加了一些 method:

- next_cp – 取得下一個控制點的 index
- prev_cp – 取得前一個控制點的 index
- calc_pos – 計算參數空間中的 U，所對應的點的位置
- list_points – 列出一長串的點
- set_spline – 設定參數曲線的種類，並更新內部的 Arc_Len_Accum
- set_tension – 設定 Cardinal Curve 的 Tension
- T_to_S – 將參數空間中的 T 換成實際距離 S
- S_to_T – 將實際距離 S 換成參數空間中的 T
- Arc_Len_Accum – 取得軌道的曲線長累積表

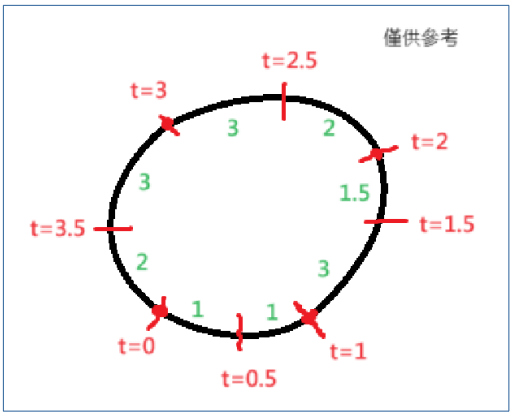
TOP

四資工二甲　B11132021　蔣博元

演算法

Arc Length Parameterization

首先需要建立 **t** 和 **s** 間的對應關係，我將這對應關係存在 Arc_Len_Accum，會這樣命名是因為表中的 **s** 是曲線長不斷累加的結果。



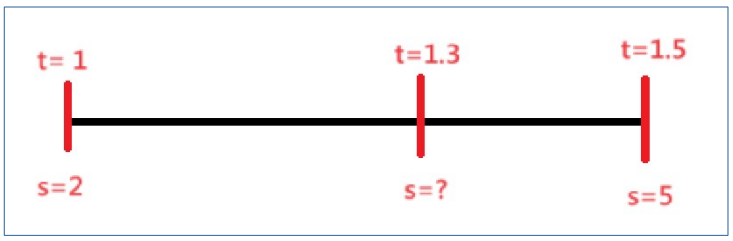
假設我的軌道長這樣，有四個控制點在 **t = 0, 1, 2, 3** 的點上，然後綠色表示那一段的曲線長。如果我的 **t** 是每 0.5 取一次，那麼我會建出如下的對應關係：

t	0	0.5	1	1.5	2	2.5	3	3.5	4
s	0	1	2	5	6.5	8.5	11.5	14.5	16.5

在這表中 **t = 4** 和 **t = 0** 是同一點，而 **t = 4** 對應的 **s** 是整圈軌道的長度。

有了這表，我們就能用線性內插來完成 **t** 和 **s** 間的轉換。

假設要將 **t = 1.3** 轉成 **s**，因為 1.3 在 1 和 1.5 間，所以會取 **s = 2** 和 **s = 5** 做線性內插，得到 **t=1.3** 對應的 **s** 大約是 $\frac{0.3}{0.5}*5+\frac{0.2}{0.5}*2=3.8$。



要將 **s** 轉成 **t** 也是類似的做法。

【註】在建表格時，t 要取得夠多，線性內插的結果才會比較準。

TOP

軌道繪製

軌道雖然是曲線，但只要將軌道拆成夠多段每一段都用直線繪製，還是能畫出曲線的效果。

當 ArcLen 關閉時，軌道是以固定的 t 為間隔去分段，不過 t 的間隔固定不代表實際距離 s 的間隔也會固定，因此會造成每段軌道的長度長短不一。

當 ArcLen 開啟時，軌道則是以固定的 s 去分段。因為我在計算曲線上點的位置時需要 t，所以可以用上一頁提到的方法將 s 轉成 t 再求點的位置。

軌道上某一點的座標可以用參數式來求。同樣，那一點的 orient 也能用參數式來求。

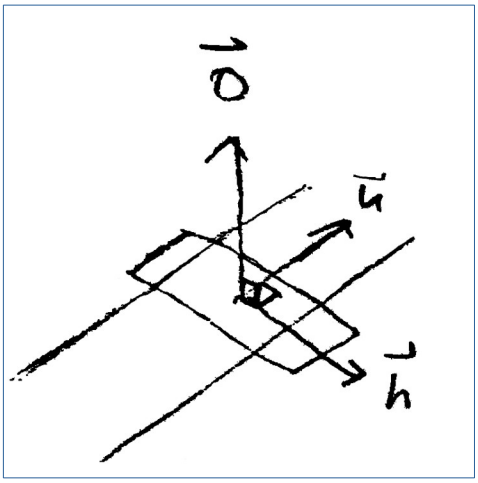
我的參數式是以 P(t) = G M T 的型式表示。

計算點座標時 G 會是控制點的座標組成的矩陣，要計算 orient 時將 G 改成控制點的 orient 就行了。

---

orient 是軌道朝向的方向但不一定是法向量。

下圖是一段斜斜向上的軌道：向量 o 是 orient 指向天空、向量 u 是平行軌道方向的方向向量，先算向量 u 和向量 o 的外積會得到軌道面上的另一個向量 h，之後再算向量 h 和向量 u 的外積就能得到軌道面真正的法向量。



TOP

Shader

Shader 是在 GPU 上運行的小程式。新版的 OpenGL 偏好使用 Shader 和 VAO 來取代 glBegin、glVertex 等 API。本次專案實作了 Vertex Shader 和 Fragment Shader。

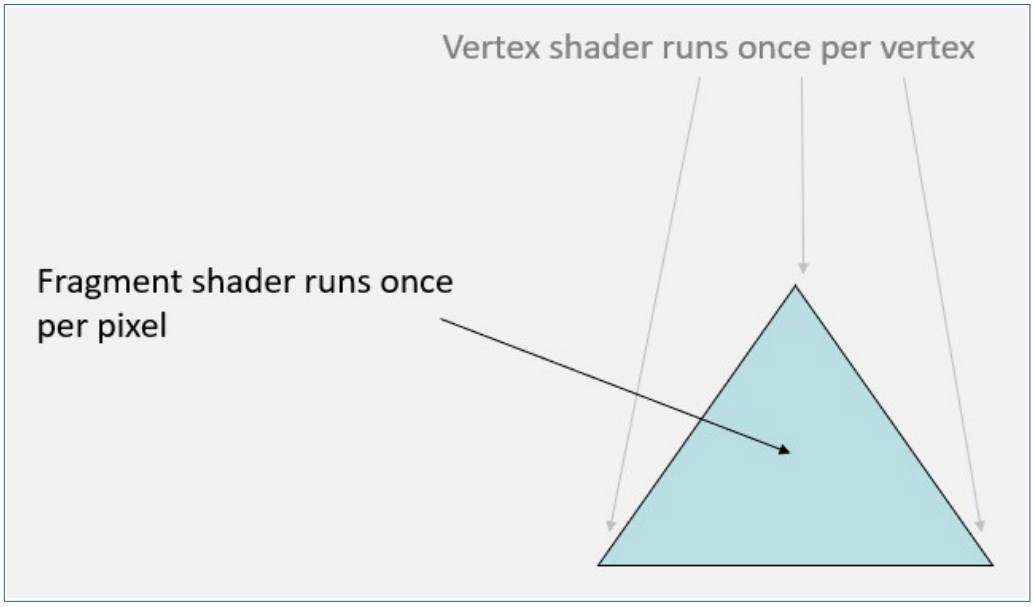
Vertex Shader 可以用來改變頂點的位置，就像模擬 sine wave 時做的一樣。

它的輸入稱作 Attribute，在 shader 內會有 `layout(location = n) in …` 來表示一個 Attribute，其中 location 指定了它的 index，這在呼叫 glVertexAttribPointer 會用到。

它的輸出為頂點在 clip coordinate 的座標，所以需要將 projection matrix、view matrix、model matrix 等傳入 shader 中才能做計算。該輸出要寫入 gl_Position。

Fragment Shader 則是決定每一像素點的顏色。它有一個輸出（可以把它命名成 `out vec4 Color;`），計算好的顏色要存進這個輸出內。

OpenGL 會針對圖形的每個頂點執行一次 Vertex Shader，之後用內插得出圖形上每個點的位置並針對這些點各執行一次 Fragment Shader。

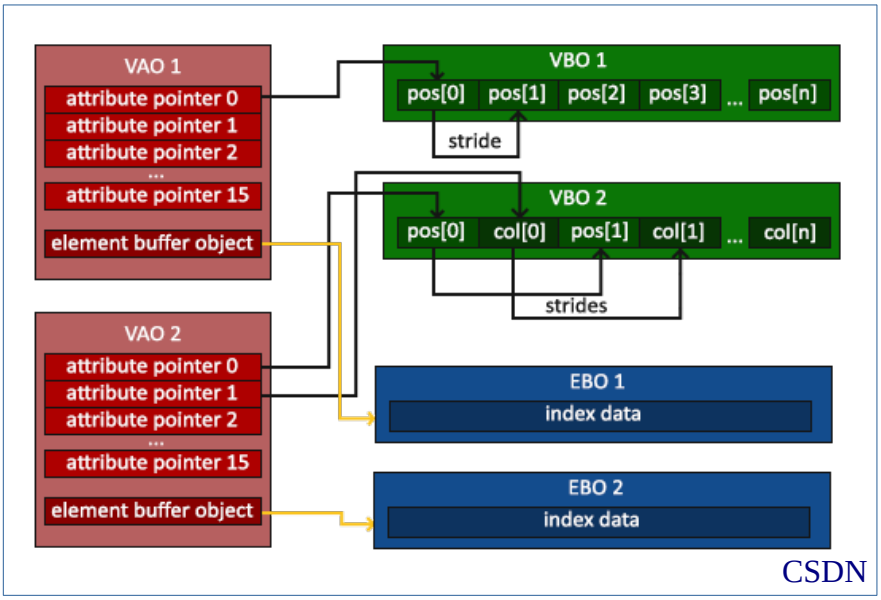


TOP

VAO

VAO（Vertex Array Object）用來將一個圖形上數個頂點的資訊整合在一起。glGenVertexArrays 用來產生 VAO，glBindVertexArray 用來綁定 VAO。

在綁定 VAO 後，VAO 將能記下接下來綁定的 VBO 和 EBO，這樣之後重新綁定 VAO 時 VBO 和 EBO 也會跟著綁定。



VBO 儲存了頂點的 Attribute。大致的設定方式如下：

```
glGenBuffers(…);                    // 產生 VAO
glBindBuffer(GL_ARRAY_BUFFER, …); // 綁定
glBufferData(…);                    // 將資料傳進 VBO
glVertexAttribPointer(n, …);        // 綁定到特定的 index
glEnableVertexAttribArray(n);       // 啟用該 index
```

上面的 index n 就是 Vertex Shader 中 layout (location = n) in …替 Attribute 設定的 index。綁定到特定 index 後，就能用這組 VBO 來傳遞 Attribute。

EBO 記錄了頂點的 index（不是上面講的 Attribute 的 index），有了 EBO 就可以用 glDrawElements 這個 API 來從 EBO 讀出頂點的 index 並繪製相對應的頂點。大致設定方式如下：

```
glGenBuffers(…);                              // 產生 EBO
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, …);  // 綁定
glBufferData(…);                              // 將資料傳進 EBO
```

TOP

四資工二甲　B11132021　蔣博元

Uniform 和 UBO

uniform 類似於 shader 中的全域變數，可以透過 glUniform 這個 API 來將單一的 uniform 傳入 shader 內。或者也能用 UBO 來傳遞一組的 uniform。

要使用 UBO 需要在 shader 內宣告一個 uniform block，例如：

```
layout (std140, binding = 0) uniform Matrices
{
   mat4 projection;
   mat4 view;
};
```

std140 指定了 memory layout，而 binding 指定了 binding point。當 uniform block 包含許多不同的資料型態時可能會出現 padding byte 的問題，而 std140 提供了一個標準來記算 padding byte。

在 C++中設定 UBO 的方式大致如下：

```
glGenBuffers(…);                                  // 產生 UBO
glBindBuffer(GL_UNIFORM_BUFFER, …);               // 綁定
glBufferData(…);                                  // 傳資料進 UBO
glBindBufferBase(GL_UNIFORM_BUFFER, binding, …);  // 綁定到特定 binding point
```

glBindBuffer 主要是為了設定 UBO 的資料才綁定的，而 glBindBufferBase（或 glBindBufferRange）才是真正的將 UBO 綁定到 shader 能取用的 binding point。

因為 uniform block 通常會有一個以上的資料，所以可以用 glBufferSubData 來修改一部份的 UBO。

Reference: Learn OpenGL - Advanced GLSL - Uniform Buffer Object

TOP