# 組合語言 書面報告

**第六組 報告順位37**

## Table of Content



## 使用函式庫

### 系統自帶

```
include macros/utils.mac
include macros/int.mac
include macros/float.mac
include macros/vector.mac
```

### 額外下載

無

## 遊戲介紹

《菜機》是一款純粹由X86組合語言製作的遊戲，玩家將扮演飛機駕駛員，在遊戲中駕駛飛機穿越融合了高等線性代數的宇宙障礙物堆。

在遊戲中，玩家將需要掌握各種飛行技巧，才能夠成功穿越障礙物堆並到達目的地。遊戲中的隕石堆是由多個獨立的障礙物堆組成，並且會不斷移動，因此玩家需要根據當前障礙物堆的位置和移動路徑，來決定飛行軌跡。

除了穿越障礙物堆的挑戰之外，遊戲中還設有多種不同的功能，包括切換遊戲視角等等。

《菜機》是一款非常具有挑戰性的遊戲，玩家將需要應用自己的飛行技巧和高等線性代數知識，才能夠成功穿越障礙物堆。同時，遊戲中的多樣化視角及難到爆炸的原始碼，也讓遊戲充滿了許多可玩性。

綜合來說，《菜機》是一款以駕駛飛機穿越宇宙障礙物堆為主題的遊戲，透過融合了高等線性代數的遊戲技術，為玩家提供了一個充滿挑戰和娛樂的遊戲體驗。

# 遊戲函式介紹

(以下都已pseudocode呈現)

## Camera類

### Distortion - 子類別

```
class Distortion:
    def init(Vec3 pos, float siz, int type):
        Vec3 self.pos = pos
        float self.siz = siz
        float self.r = 50
        float self.k = 1.5
        int self.type = type

        if self.type == 1:
            self.r = self.siz + self.siz ** self.k
        endif
    endf

    def need(Vec3 p) -> (int):
        if self.type == 0:
            return 0
        elif self.type == 1:
            Vec3 v = self.pos - p
            return v.length() < self.r
        elif self.type == 2:
            return fabs((p - self.pos).length() - self.r) < self.siz
        elif self.type == 3:
            return fabs(self.pos.x - p.x) < self.siz
        endif
    endf

    def update():
        if self.type == 1:
            float nx = max2f(player.pos.x - 100 - self.r, self.pos.x + 0.9 *
player.max_speed * engine.dt)
            self.pos = Vec3(nx, player.pos.y, 40)
        elif self.type == 2:
            self.r = self.r + 10 * engine.dt
            if self.r > 500:
                self.type = 0
            endif
        elif self.type == 3:
            self.pos.x = self.pos.x + 200 * engine.dt
            if self.pos.x - player.pos.x > 1000:
                self.type = 0
            endif
```

```
        endif
    endf

    def distort(Vec3 p) -> (Vec3):
        if self.type == 1:
            Vec3 v = self.pos - p
            float d = v.length()
            float x = d - self.siz ** self.k
            if 0 < x && x < self.siz:
                return v.mulc(((self.siz - x) ** self.k) / d)
            else:
                return self.pos - p
            endif
        elif self.type == 2:
            Vec3 v = p - self.pos
            float d = v.length()
            float x = d - self.r
            if fabs(x) < self.siz:
                float k = 0.7
                float t = (self.siz - ((self.siz * cos(x / 2 / self.siz *
3.1415926)) ** k) / (self.siz ** (k - 1))) * x / fabs(x)
                return v.mulc((t - x) / d)
            else:
                return Vec3_zero()
            endif
        elif self.type == 3:
            float x = self.pos.x - p.x
            if fabs(x) < self.siz:
                float h = 30
                float t = (1 + cos(x * 3.1415926 / self.siz)) * h / 2
                return Vec3(0, 0, t)
            else:
                return Vec3_zero()
            endif
        endif
    endf
endc
```

- init 方法會被呼叫來建立新的 Distortion 物件，接收三個參數：pos、siz 和 type。其中，pos 是 Vec3 類型的位置，siz 是浮點數的大小，type 是整數的類型。init 方法會將 pos、siz 和 type 分別存入新建立的 Distortion 物件的 pos、siz 和 type 屬性中。
- need 方法會接收一個參數 p（型態為 Vec3），並傳回一個整數。根據 Distortion 物件的 type 屬性的值，need 方法會回傳一個判斷式的結果。
- update 方法會根據 Distortion 物件的 type 屬性的值，來更新 Distortion 物件的其他屬性值。
- distort 方法會接收一個參數 p（型態為 Vec3），並傳回一個 Vec3 類型的結果。根據 Distortion 物件的 type 屬性的值，distort 方法會計算並傳回一個新的 Vec3 向量。

**Camera - 子類別**

**init**

```
def init():
    Vec3 self.pos = Vec3(-100, 0, 0)
    Vec3 self.near = Vec3(40, 30, 40)
    Vec3 self.near_gp
    Matrix3 self.axis
    Matrix3 self.inv_axis
    self.set_axis(Matrix3(Vec3(0, 0, -1), Vec3(-1, 0, 0), Vec3(0, -1, 0)))
    Vec3 self.canvas_offset = Vec3(WIDTH / 2, HEIGHT / 2, 0)
    Vec3 self.canvas_scale = Vec3(WIDTH / self.near.x, HEIGHT / self.near.y, 1)

    Vec3 self.sun_dir = Vec3(1, 0, 1).norm()
    float self.env_light = 0.2
    float self.diffuse = 0.4
    float self.bloom = 0.4

    float self.mesh_size = 0.01

    float self.shake_start_time = 0
    float self.shake_magnitude = 0
    float self.shake_duration = 0

    int self.cnt = 0

    int self.n_distortions = 10
    Distortion self.distortions[10]
    for i in range(self.n_distortions):
        self.distortions[i].type = 0
    endl
endf
```

這段程式碼是在定義一個 init 函數，在這個函數裡面初始化兩個向量 self.pos 和 self.near、三個矩陣 self.axis、self.inv_axis 和 self.canvas_scale、浮點數變量：self.env_light、self.diffuse 和 self.bloom、一個整數變量 self.cnt 和一個存儲了 Distortion 類型對象的數組 self.distortions還有一些浮點數變量：self.shake_start_time、self.shake_magnitude 和 self.shake_duration。

**add_distortion**

```
def add_distortion(Distortion d):
    for i in range(self.n_distortions):
        if self.distortions[i].type == 0:
            self.distortions[i] = d
            return
        endif
    endl
endf
```

個函數的作用是在 self.distortions 數組中添加一個新的 Distortion 對象。

**need_distort**

```
def need_distort(Vec3 p) -> (int):
    for i in range(self.n_distortions):
        if self.distortions[i].need(p) == 1: return 1
    endl
    return 0
endf
```

判斷向量 p 是否需要扭曲。如果需要，就返回 1；否則返回 0。

**distort**

```
def distort(Vec3 p) -> (Vec3):
    Vec3 r = p
    for i in range(self.n_distortions):
        if self.distortions[i].type > 0:
            r = r + self.distortions[i].distort(p)
        endif
    endl
    return r
endf
```

對向量 p 進行扭曲。它會依次調用 self.distortions 數組中所有元素的 distort 方法，並累加它們的返回值。

**set_axis**

```
def set_axis(Matrix3 axis):
    self.axis = axis
    self.inv_axis = axis.inv()
    self.near_gp = self.pos + self.inv_axis.w.mulc(self.near.z)
endf
```

設置幾個成員變量的值。

**set_inv_axis**

```
def set_inv_axis(Matrix3 inv_axis):
    self.inv_axis = inv_axis
    self.axis = inv_axis.inv()
    self.near_gp = self.pos + self.inv_axis.w.mulc(self.near.z)
endf
```

設置幾個成員變量的值。

**render**

```
def render():
    self._shake()
    for i in range(self.n_distortions):
        self.distortions[i].update()
    endl
    clear_buffer()
    for i in range(n_triangles):
        Triangle tri = triangles[i]
        Vec3 p0 = vertices[tri.p0]
        Vec3 p1 = vertices[tri.p1]
        Vec3 p2 = vertices[tri.p2]

        Vec3 bound = self.distortions[0].pos
        float bound_r = self.distortions[0].r - self.distortions[0].siz
        float bound_x = bound.x
        if (p0.x < bound_x || (p0 - bound).length() < bound_r) && \
            (p1.x < bound_x || (p1 - bound).length() < bound_r) && \
            (p2.x < bound_x || (p2 - bound).length() < bound_r):
            continue
        endif

        ; clip near
        if self.visible(p0) == 0 && self.visible(p1) == 0 && self.visible(p2) ==
0:
            continue
        endif

        float l0 = (p1 - p2).length()
        float l1 = (p0 - p2).length()
        float l2 = (p0 - p1).length()

        ; int x0 = self.need_distort(p0)
        ; int x1 = self.need_distort(p1)
        ; int x2 = self.need_distort(p2)

        if self.need_distort(p0) == 1 || self.need_distort(p1) == 1 ||
self.need_distort(p2):
            if l0 > l1 && l0 > l2:
                self.seperate_triangle(p0, p1, p2)
            elif l1 > l2:
                self.seperate_triangle(p1, p2, p0)
            else:
                self.seperate_triangle(p2, p0, p1)
            endif
        else:
            self.render_triangle(p2, p1, p0)
        endif
```

```
    endl
endf
```

實現一個畫圖的功能。它會從 triangles 這個陣列裡面取出每一個三角形，並進行一些判斷。如果三角形的頂點超出畫面範圍或是在畫面背景前方，就不進行畫圖。如果三角形的頂點在畫面背景後方，就根據三角形各頂點之間的距離，選擇最長的那條邊進行分割，並進行畫圖。

**seperate_triangle**

```
def seperate_triangle(Vec3 p0, Vec3 p1, Vec3 p2):
    Vec3 p = (p1 + p2).mulc(0.5)
    if (p1 - p2).length() < (p - self.pos).length() * self.mesh_size:
        self.render_triangle(self.distort(p2), self.distort(p1), self.distort(p0))
    else:
        self.seperate_triangle(p, p0, p1)
        self.seperate_triangle(p, p2, p0)
    endif
endf
```

將傳入的三角形分割成兩個小三角形。它會先計算三角形的中心點 p，然後比較 p 與三角形頂點之間的距離是否小於三角形邊長的一半。如果是，就直接畫出傳入的三角形。如果不是，則將傳入的三角形分割成兩個小三角形，分別以 p 為頂點，並遞归呼叫自己。

**visible**

```
def visible(Vec3 p) -> (int):
    float t
    Vec3 gp
    t, gp = rayPlaneIntersect(self.pos, p, self.near_gp, self.inv_axis.w)
    Vec3 cp = self.inv_axis.transform(gp - self.near_gp) * self.canvas_scale +
self.canvas_offset
    int x = cp.x
    int y = cp.y
    return x >= 0 && x < WIDTH && y >= 0 && y < HEIGHT && t > 0
endf
```

判斷傳入的點是否在畫面範圍內。它會計算傳入的點與畫面背景平面的交點，並將交點轉換成畫面座標系中的坐標。接著判斷轉換後的坐標是否超出畫面範圍，以及傳入的點是否在畫面背景前方。如果是，則傳回 1，否則傳回 0。

**shake**

```
def shake(float magnitude, float duration):
    self.shake_start_time = engine.time
    self.shake_magnitude = magnitude
```

```
    self.shake_duration = duration
endf
```

設定震動的參數。它會記錄下震動的開始時間，並將震動的幅度和持續時間存到變數中。這些變數可以在其他函數中使用，例如在畫圖的過程中進行平移操作，以實現震動的效果。

**_shake**

```
def _shake():
    float t = engine.time - self.shake_start_time
    if t <= self.shake_duration:
        float k = self.shake_magnitude * t / self.shake_duration
        self.pos = self.pos + Vec3(rand(0-k, k), rand(0-k, k), rand(0-k, k))
    endif
endf
```

在震動持續時間內，將畫面進行隨機平移。它會計算當前的時間是否小於震動的持續時間。如果是，則根據震動的幅度和持續時間，計算出一個縮放因子 k，並將畫面隨機平移 k 倍的距離。這樣就能實現震動的效果。

**_shake**

```
def render_triangle(Vec3 v1, Vec3 v2, Vec3 v3):
    self.cnt = self.cnt + 3

    float t1, t2, t3
    Vec3 gp1, gp2, gp3

    t1, gp1 = rayPlaneIntersect(self.pos, v1, self.near_gp, self.inv_axis.w)
    t2, gp2 = rayPlaneIntersect(self.pos, v2, self.near_gp, self.inv_axis.w)
    t3, gp3 = rayPlaneIntersect(self.pos, v3, self.near_gp, self.inv_axis.w)

    if t1 < 0 || t2 < 0 || t3 < 0: return

    Vec3 N = Vec3.cross(v2 - v1, v3 - v1).norm()

    float d1 = (v1 - self.pos).length()
    float d2 = (v2 - self.pos).length()
    float d3 = (v3 - self.pos).length()

    Vec3 p1 = self.inv_axis.transform(gp1 - self.near_gp) * self.canvas_scale +
self.canvas_offset
    Vec3 p2 = self.inv_axis.transform(gp2 - self.near_gp) * self.canvas_scale +
self.canvas_offset
    Vec3 p3 = self.inv_axis.transform(gp3 - self.near_gp) * self.canvas_scale +
self.canvas_offset

    float wf = (p2.y-p3.y) * (p1.x-p3.x) + (p3.x-p2.x) * (p1.y-p3.y)
```

```
; calculate light
; Vec3 h = (self.sun_dir + ).norm()
float _light = self.env_light + self.diffuse * max2f(0, N.dot(self.sun_dir))

; N.print()
; printFloat light
; printInt color


; fixed-point coordinates
int X1 = 16 * p1.x
int X2 = 16 * p2.x
int X3 = 16 * p3.x

int Y1 = 16 * p1.y
int Y2 = 16 * p2.y
int Y3 = 16 * p3.y

; Deltas
int DX12 = X1 - X2
int DX23 = X2 - X3
int DX31 = X3 - X1

int DY12 = Y1 - Y2
int DY23 = Y2 - Y3
int DY31 = Y3 - Y1

; Fixed-point deltas
int FDX12 = DX12 << 4
int FDX23 = DX23 << 4
int FDX31 = DX31 << 4

int FDY12 = DY12 << 4
int FDY23 = DY23 << 4
int FDY31 = DY31 << 4

; Bounding rectangle
int minx = (min3i(X1, X2, X3) + 15) >> 4
int maxx = (max3i(X1, X2, X3) + 15) >> 4
int miny = (min3i(Y1, Y2, Y3) + 15) >> 4
int maxy = (max3i(Y1, Y2, Y3) + 15) >> 4

minx = max2i(minx, 0)
maxx = min2i(maxx+1, WIDTH)
miny = max2i(miny, 0)
maxy = min2i(maxy+1, HEIGHT)

; Half-edge constants
int C1 = DY12 * X1 - DX12 * Y1;
int C2 = DY23 * X2 - DX23 * Y2;
int C3 = DY31 * X3 - DX31 * Y3;

; Correct for fill convention
if DY12 < 0 || (DY12 == 0 && DX12 > 0): C1 = C1 + 1
```

```
    if DY23 < 0 || (DY23 == 0 && DX23 > 0): C2 = C2 + 1
    if DY31 < 0 || (DY31 == 0 && DX31 > 0): C3 = C3 + 1

    int CY2 = C2 + DX23 * (miny << 4) - DY23 * (minx << 4)
    int CY1 = C1 + DX12 * (miny << 4) - DY12 * (minx << 4)
    int CY3 = C3 + DX31 * (miny << 4) - DY31 * (minx << 4)

    for y in range(miny, maxy):
        int CX1 = CY1
        int CX2 = CY2
        int CX3 = CY3

        for x in range(minx, maxx):
            if CX1 > 0 && CX2 > 0 && CX3 > 0:
                float w1, w2, w3, d
                w1 = ((p2.y-p3.y) * (x-p3.x) + (p3.x-p2.x) * (y-p3.y)) / wf
                w2 = ((p3.y-p1.y) * (x-p3.x) + (p1.x-p3.x) * (y-p3.y)) / wf
                w3 = 1 - w1 - w2
                d = w1 * d1 + w2 * d2 + w3 * d3

                if d < deep_buffer[y][x]:
                    ; draw pixel
                    Vec3 v = Vec3(w1*v1.x + w2*v2.x + w3*v3.x, w1*v1.y + w2*v2.y +
w3*v3.y, w1*v1.z + w2*v2.z + w3*v3.z).norm()
                    Vec3 h = (self.sun_dir + v).norm()
                    float light = _light + max2f(0, N.dot(h)) * self.bloom
                    int color = light * 64
                    color_buffer[y][x] = color
                    deep_buffer[y][x] = d
                endif
            endif

            CX1 = CX1 - FDY12
            CX2 = CX2 - FDY23
            CX3 = CX3 - FDY31
        endl

        CY1 = CY1 + FDX12
        CY2 = CY2 + FDX23
        CY3 = CY3 + FDX31

    endl

endf
```

將傳入的三角形畫在畫面上。它會先計算傳入的三角形的頂點與畫面背景平面的交點，並將交點轉換成畫面座標系中的坐標。接著計算出三角形的法向量，並計算出各頂點與觀察者的距離。最後，根據傳入的三角形的頂點坐標，計算出畫面上每個像素對應的三角形頂點坐標，並根據傳入的三角形的法向量和太陽方向計算出燈光強度，最後將三角形繪製到畫面上。

## Engine類

**get_time**

```
def get_time() -> (int):
    int t
    call GetMseconds
    mov t, eax
    return t
endf
```

函式用來取得當前系統時間，並以毫秒為單位返回。

**Keyboard**

```
class Keyboard:
    def init():
        int self.q
        int self.e
        int self.w
        int self.a
        int self.s
        int self.d
        int self.up = 0
        int self.down = 0
        int self.left = 0
        int self.right = 0
        int self.esc = 0
    endf

    def update():
        call ReadKey
        int key
        movzx ebx, ax
        mov key, ebx
        self.q      = key == 04209
        self.e      = key == 04709
        self.w      = key == 04471
        self.a      = key == 07777
        self.s      = key == 08051
        self.d      = key == 08292
        self.up     = key == 18432
        self.down   = key == 20480
        self.left   = key == 19200
        self.right  = key == 19712
        self.esc    = key == 00283
    endf

    def print():
        printInt self.up
        printInt self.down
        printInt self.left
```

```
        printInt self.right
        printInt self.esc
        printEndl
    endf
endc
```

用來管理鍵盤輸入。它有一些變數用來儲存各個鍵是否被按下，以及一個 update 函式用來更新鍵盤輸入的狀態。

**Engine**

```
class Engine:
    def init():
        float self.update_rate = 16
        float self.dt
        int self.last_t = get_time()
    endf

    def step():
        float t = get_time()
        while t - self.last_t < self.update_rate:
            t = get_time()
        endl
        self.dt = (t - self.last_t) / 1000.0
        ; printFloat self.dt
        self.last_t = t

        mm.reset()
        update()
        keyboard.update()
        camera.render()
        display()
    endf
endc
```

用來管理遊戲引擎的更新流程。它有一個 step 函式用來控制每個更新週期的持續時間，然後調用更新函式、鍵盤輸入函式和渲染函式。

**main**

```
def main():
    build_char_level()

    camera = Camera()
    engine = Engine()
    keyboard = Keyboard()
    mm = ModelManager()
```

```
    init()
    while True:
        engine.step()
    endl
    ; testRayTriangleIntersect()
    ; testRender()
    ; testRenderAnimation()
    ; testRenderTriangle()
    call WaitMsg
endf
```

是程式的主入口，它會建立遊戲引擎、鍵盤和模型管理器的實例，然後在無限迴圈中調用引擎的更新函式。

## math類

#### fsqrt

```
def fsqrt(float x) -> (float):
    ffsqrt x, x
    return x
endf
```

這是一個用來計算平方根的函數。它使用了 ffsqrt 指令將輸入的浮點數 x 的平方根存回 x 中，然後將結果傳回給呼叫函數的程式。

#### fabs

```
def fabs(float x) -> (float):
    ffabs x, x
    return x
endf
```

這是一個用來計算絕對值的函數。它使用了 ffabs 指令將輸入的浮點數 x 的絕對值存回 x 中，然後將結果傳回給呼叫函數的程式。

#### sin

```
def sin(float x) -> (float):
    ffld x
    fsin
    fstp x
    return x
endf
```

這是一個用來計算正弦值的函數。它使用了 ffld 指令將輸入的浮點數 x 存入浮點數堆棧中，然後使用了 fsin 指令將堆棧頂端的數字轉換為其正弦值。接著使用了 fstp 指令將結果存回 x 中，最後將結果傳回給呼叫函數的程式。

**cos**

```
def cos(float x) -> (float):
    ffld x
    fcos
    fstp x
    return x
endf
```

這是一個用來計算餘弦值的函數。

**round**

```
def round(float x) -> (int):
    ffld x
    fstcw fcw
    or fcw, 0000110000000000b
    fldcw fcw
    int y
    fistp y
    return y
endf
```

這是一個用來將浮點數四捨五入為最接近的整數的函數。它使用了 ffld 指令將輸入的浮點數 x 存入浮點數堆棧中，然後使用了 fstcw 指令將當前浮點數控制字存入記憶體位址 fcw 中。接著使用了 or 指令將 fcw 的內容與二進位數值 0000110000000000b 做邏輯或運算，這會將浮點數控制字的最後兩個位元設為 11，這會使得浮點數四捨五入功能被啟用。接著使用了 fldcw 指令將修改後的浮點數控制字載入處理器，最後使用了 fistp 指令將堆棧頂端的數字轉換為整數並存回變數 y 中。最後將結果傳回給呼叫函數的程式。

**min2i**

```
def min2i(int x, int y) -> (int):
    if x < y: return x
    return y
endf
```

這是一個用來比較兩個整數並傳回較小者的函數。它使用了條件運算子 if 來檢查變數 x 是否小於變數 y，如果是，則傳回 x。

**max2i**

```
def max2i(int x, int y) -> (int):
    if x > y: return x
    return y
endf
```

這是一個用來比較兩個整數並傳回較大者的函數。它使用了條件運算子 if 來檢查變數 x 是否大於變數 y，如果是，則傳回 x，否則傳回 y。

**min2f**

```
def min2f(float x, float y) -> (float):
    if x < y: return x
    return y
endf
```

這是一個用來比較兩個浮點數並傳回較小者的函數。它使用了條件運算子 if 來檢查變數 x 是否小於變數 y，如果是，則傳回 x，否則傳回 y。

**max2f**

```
def max2f(float x, float y) -> (float):
    if x > y: return x
    return y
endf
```

這是一個用來比較兩個浮點數並傳回較大者的函數。它使用了條件運算子 if 來檢查變數 x 是否大於變數 y，如果是，則傳回 x，否則傳回 y。

**min3i**

```
def min3i(int x, int y, int z) -> (int):
    if x < y && x < z:
        return x
    elif y < z:
        return y
    else:
        return z
    endif
endf
```

這是一個用來比較三個整數並傳回最小者的函數。它使用了條件運算子 if 來檢查變數 x 是否大於變數 y，如果是，則傳回 x，否則使用了條件運算子 elif 來檢查變數 y 是否大於變數 z，如果是，則傳回 y，否則傳回 z。

**min5f**

```
def min5f(float a, float b, float c, float d, float e) -> (float):
    if a < b && a < c && a < d && a < e:
        return a
    elif b < c && b < d && b < e:
        return b
    elif c < d && c < e:
        return c
    elif d < e:
        return d
    else:
        return e
endf
```

這是一個用來比較五個浮點數並傳回最小者的函數。它使用了條件運算子 if 來檢查變數 a 是否小於變數 b 且也小於變數 c、d 和 e，如果是，則傳回 a；否則使用條件運算子 elif 來檢查變數 b 是否小於變數 c、d 和 e，如果是，則傳回 b；否則使用 elif 來檢查變數 c 是否小於變數 d 和 e，如果是，則傳回 c；否則使用 elif 來檢查變數 d 是否小於變數 e，如果是，則傳回 d；否則使用 else 指令傳回 e。

**Vec2 - 子類別**

```
class Vec2:
    def init(float x, float y):
        float self.x = x
        float self.y = y
    endf

    def add(Vec2 other) -> (Vec2):
        return Vec2(self.x + other.x, self.y + other.y)
    endf

    def sub(Vec2 other) -> (Vec2):
        return Vec2(self.x - other.x, self.y - other.y)
    endf

    def mulc(float other) -> (Vec2):
        return Vec2(self.x * other, self.y * other)
    endf

    def length() -> (float):
        return fsqrt(self.x * self.x + self.y * self.y)
    endf

    def dot(Vec2 other) -> (float):
        return self.x * other.x + self.y * other.y
```

```
    endf

    def norm() -> (Vec2):
        float l = self.length()
        return Vec2(self.x / l, self.y / l)
    endf

    def print():
        printFloat self.x
        printFloat self.y
        printEndl
    endf
endc
```

- init：建構函數，用來建立新的向量物件。
- add：加法運算子，用來計算兩個向量的和。
- sub：減法運算子，用來計算兩個向量的差。
- mulc：乘法運算子，用來將向量與一個浮點數相乘。
- length：計算向量的長度。
- dot：計算向量的點積。
- norm：計算向量的正規化向量。
- print：將向量輸出至標準輸出。

**Vec3 - 子類別**

```
class Vec3:
    def init(float x, float y, float z):
        float self.x = x
        float self.y = y
        float self.z = z
    endf

    def add(Vec3 other) -> (Vec3):
        return Vec3(self.x + other.x, self.y + other.y, self.z + other.z)
    endf

    def sub(Vec3 other) -> (Vec3):
        return Vec3(self.x - other.x, self.y - other.y, self.z - other.z)
    endf

    def mul(Vec3 other) -> (Vec3):
        return Vec3(self.x * other.x, self.y * other.y, self.z * other.z)
    endf

    def mulc(float other) -> (Vec3):
        return Vec3(self.x * other, self.y * other, self.z * other)
    endf

    def length() -> (float):
```

```
        return fsqrt(self.x * self.x + self.y * self.y + self.z * self.z)
    endf

    def dot(Vec3 other) -> (float):
        return self.x * other.x + self.y * other.y + self.z * other.z
    endf

    def cross(Vec3 other) -> (Vec3):
        return Vec3(self.y * other.z - other.y * self.z, other.x * self.z - self.x
* other.z, self.x * other.y - other.x * self.y)
    endf

    def toVec2() -> (Vec2):
        return Vec2(self.x, self.y)
    endf

    def norm() -> (Vec3):
        float l = self.length()
        return Vec3(self.x / l, self.y / l, self.z / l)
    endf

    def print():
        printFloat self.x
        printFloat self.y
        printFloat self.z
        printEndl
    endf
endc
```

- init: 是類別的建構子，用於在創建物件時初始化該物件的狀態。
- add: 傳回兩個 Vec3 物件的 x、y、z 座標之和。
- sub: 傳回兩個 Vec3 物件的 x、y、z 座標之差。
- mul: 傳回兩個 Vec3 物件的 x、y、z 座標的乘積。
- mulc: 傳回自己的 x、y、z 座標乘上一個浮點數的結果。
- length: 傳回自己的長度。
- dot: 傳回兩個 Vec3 物件的內積。
- cross: 傳回兩個 Vec3 物件的外積。
- toVec2: 傳回一個 Vec2 物件，其中包含自己的 x 和 y 座標。
- norm: 傳回一個長度為 1 的 Vec3 物件，其中包含自己的 x、y、z 座標。
- print: 在控制台輸出自己的 x、y、z 座標。

#### Vec3_zero

```
def Vec3_zero() -> (Vec3):
    return Vec3(0, 0, 0)
endf
```

回傳一個 (0, 0, 0) 的 3D 向量。

**Vec3_one**

```
def Vec3_one() -> (Vec3):
    return Vec3(1, 1, 1)
endf
```

回傳一個 (1, 1, 1) 的 3D 向量。

**Matrix3 - 子類別**

```
class Matrix3:
    ; u, v, w are rows
    def init(Vec3 u, Vec3 v, Vec3 w):
        Vec3 self.u = u
        Vec3 self.v = v
        Vec3 self.w = w
    endf

    def transform(Vec3 other) -> (Vec3):
        return Vec3(self.u.dot(other), self.v.dot(other), self.w.dot(other))
    endf

    def transpose() -> (Matrix3):
        return Matrix3(Vec3(self.u.x, self.v.x, self.w.x), Vec3(self.u.y,
self.v.y, self.w.y), Vec3(self.u.z, self.v.z, self.w.z))
    endf

    def mul(Matrix3 other) -> (Matrix3):
        Matrix3 ot = other.transpose()
        return Matrix3(\
            Vec3(Vec3.dot(self.u, ot.u), Vec3.dot(self.u, ot.v), Vec3.dot(self.u,
ot.w)),\
            Vec3(Vec3.dot(self.v, ot.u), Vec3.dot(self.v, ot.v), Vec3.dot(self.v,
ot.w)),\
            Vec3(Vec3.dot(self.w, ot.u), Vec3.dot(self.w, ot.v), Vec3.dot(self.w,
ot.w))\
        )
    endf

    def inv() -> (Matrix3):
        float det = self.u.x * (self.v.y * self.w.z - self.w.y * self.v.z) - \
                    self.u.y * (self.v.x * self.w.z - self.v.z * self.w.x) + \
                    self.u.z * (self.v.x * self.w.y - self.v.y * self.w.x)

        float invdet = 1 / det

        Matrix3 inv
        inv.u.x = (self.v.y * self.w.z - self.w.y * self.v.z) * invdet
        inv.u.y = (self.u.z * self.w.y - self.u.y * self.w.z) * invdet
        inv.u.z = (self.u.y * self.v.z - self.u.z * self.v.y) * invdet
```

```
        inv.v.x = (self.v.z * self.w.x - self.v.x * self.w.z) * invdet
        inv.v.y = (self.u.x * self.w.z - self.u.z * self.w.x) * invdet
        inv.v.z = (self.v.x * self.u.z - self.u.x * self.v.z) * invdet
        inv.w.x = (self.v.x * self.w.y - self.w.x * self.v.y) * invdet
        inv.w.y = (self.w.x * self.u.y - self.u.x * self.w.y) * invdet
        inv.w.z = (self.u.x * self.v.y - self.v.x * self.u.y) * invdet

        return inv
    endf

    def print():
        self.u.print()
        self.v.print()
        self.w.print()
        printEndl
    endf
endc
```

- Matrix3 是一個 3x3 的矩陣，由三個 3D 向量代表三行組成。
- 建構子 init 用於建立一個 Matrix3 物件。
- 函式 transform 將矩陣與向量相乘，回傳乘積結果，也就是另一個 3D 向量。
- 函式 transpose 回傳矩陣的轉置矩陣。
- 函式 mul 將矩陣與另一個矩陣相乘，回傳乘積結果，也就是另一個 Matrix3 物件。
- 函式 inv 回傳矩陣的反矩陣。
- 函式 print 將矩陣的值輸出到螢幕。

#### Matrix3_identity

```
def Matrix3_identity() -> (Matrix3):
    return Matrix3(Vec3(1, 0, 0), Vec3(0, 1, 0), Vec3(0, 0, 1))
endf
```

回傳一個單位矩陣，也就是對角線值為 1，其他值為 0 的矩陣。

#### AngleAxis - 子類別

```
class AngleAxis:
    def init(Vec3 axis, float angle):
        Vec3 self.axis = axis
        float self.angle = angle
    endf

    def rotate(Vec3 v) -> (Vec3):
        return v.mulc(cos(self.angle)) + self.axis.cross(v).mulc(sin(self.angle))
+ self.axis.mulc(self.axis.dot(v) * (1 - cos(self.angle)))
    endf
```

```
endc
```

這是一個類別，名稱為 AngleAxis。它代表一個旋轉轉軸和角度。它包含一個函式，名稱為 init，它是建構子。建構子接收一個 3D 向量 axis 和一個浮點數 angle 作為參數，分別代表旋轉轉軸和角度。

函式 rotate 接收一個 3D 向量 v 作為參數，回傳旋轉過後的 3D 向量。

**AngleAxis - 子類別**

```
class AngleAxis:
    def init(Vec3 axis, float angle):
        Vec3 self.axis = axis
        float self.angle = angle
    endf

    def rotate(Vec3 v) -> (Vec3):
        return v.mulc(cos(self.angle)) + self.axis.cross(v).mulc(sin(self.angle))
+ self.axis.mulc(self.axis.dot(v) * (1 - cos(self.angle)))
    endf
endc
```

這是一個類別，名稱為 AngleAxis。它代表一個旋轉轉軸和角度。它包含一個函式，名稱為 init，它是建構子。建構子接收一個 3D 向量 axis 和一個浮點數 angle 作為參數，分別代表旋轉轉軸和角度。

函式 rotate 接收一個 3D 向量 v 作為參數，回傳旋轉過後的 3D 向量。

**Triangle - 子類別**

```
class Triangle:
    def init(int p0, int p1, int p2):
        int self.p0 = p0 + n_vertices
        int self.p1 = p1 + n_vertices
        int self.p2 = p2 + n_vertices
    endf
endc
```

這個函式有三個整數型態的參數 p0、p1、p2，沒有回傳值。在函式中，它會將參數 p0 加上 n_vertices 之後賦值給物件的屬性 self.p0，將參數 p1 加上 n_vertices 之後賦值給物件的屬性 self.p1，以及將參數 p2 加上 n_vertices 之後賦值給物件的屬性 self.p2。

**rayPlaneIntersect**

```

```

```
def rayPlaneIntersect(Vec3 p0, Vec3 p1, Vec3 P, Vec3 N) -> (float, Vec3):
    Vec3 dir = (p1 - p0).norm()
    float NdotRayDirection = N.dot(dir)
    if fabs(NdotRayDirection) < 0.000001:
        return INF, None
    endif

    float t = N.dot(P - p0) / NdotRayDirection
    return t, p0 + dir.mulc(t)
endf
```

這個函式有四個 Vec3 型態的參數 p0、p1、P、N，並有一個回傳值，是一個 tuple，包含一個 float 型態的變數和一個 Vec3 型態的變數。

**rayPlaneDist**

```
def rayPlaneDist(Vec3 p0, Vec3 p1, Vec3 P, Vec3 N) -> (float):
    Vec3 dir = (p1 - p0).norm()
    float NdotRayDirection = N.dot(dir)
    if fabs(NdotRayDirection) < 0.000001:
        return INF
    endif
    return N.dot(P - p0) / NdotRayDirection
endf
```

這個函式有四個 Vec3 型態的參數 p0、p1、P、N，並有一個回傳值，是一個 float 型態的變數。

函式的第一行建立了一個 Vec3 型態的變數 dir，並將 p1 和 p0 相減之後取 norm（即取該向量的方向），再將結果賦值給 dir。第二行建立了一個 float 型態的變數 NdotRayDirection，並將 N 和 dir 進行 dot 運算（即內積）之後賦值給 NdotRayDirection。接下來是一個條件判斷式，如果 NdotRayDirection 的絕對值小於 0.000001，就回傳 INF，否則執行下一行。最後一行回傳 N 和 (P - p0) 進行 dot 運算之後除以 NdotRayDirection 的結果。

**deg2rad**

```
def deg2rad(float deg) -> (float):
    return deg * 3.1415926 / 180
endf
```

float 型態的參數 deg，並有一個回傳值，是一個 float 型態的變數。 函式僅包含一行，將參數 deg 乘上 3.1415926 除以 180 之後回傳。

**axisAngle2Matrix**

```
def axisAngle2Matrix(Vec3 v, float angle) -> (Matrix3):
    angle = deg2rad(angle)

    float x = v.x
    float y = v.y
    float z = v.z

    float c = cos(angle)
    float s = sin(angle)
    float t = 1.0 - c

    return Matrix3(\
        Vec3(t*x*x + c, t*x*y - z*s, t*x*z + y*s),\
        Vec3(t*x*y + z*s, t*y*y + c, t*y*z - x*s),\
        Vec3(t*x*z - y*s, t*y*z + x*s, t*z*z + c)\
    )
endf
```

函式的第一行呼叫了 deg2rad 函式，將參數 angle 轉換成弧度值之後賦值給變數 angle。接下來是三行，分別建立了三個 float 型態的變數 x、y、z，並將 v 的 x、y、z 分別賦值給這三個變數。再來是三行，分別建立了三個 float 型態的變數 c、s、t，並將 cos(angle)、sin(angle)、1.0 - c 分別賦值給這三個變數。最後一行回傳一個 Matrix3 物件，其中包含三個 Vec3 物件。

#### rayTriangleIntersect

```
def rayTriangleIntersect(Vec3 pos, Vec3 dir, Triangle tri) -> (float, Vec3):

    Vec3 v0 = vertices[tri.p0]
    Vec3 v1 = vertices[tri.p1]
    Vec3 v2 = vertices[tri.p2]

    t, p = rayPlaneIntersect(pos, dir, v0, v1, v2)
    if t == INF: return INF, None

    Vec3 C, edge, vp

    ; edge 0
    edge = v1 - v0
    vp = p - v0
    C = edge.cross(vp)
    if N.dot(C) < 0: return INF, None

    ; edge 1
    edge = v2 - v1
    vp = p - v1
    C = edge.cross(vp)
    if N.dot(C) < 0: return INF, None

    ; edge 2
    edge = v0 - v2
```

```
    vp = p - v2
    C = edge.cross(vp)
    if N.dot(C) < 0: return INF, None

    return t, p
endf
```

這個函式有三個參數，分別是 Vec3 型態的 pos、dir 和 Triangle 型態的 tri，並有一個回傳值，是一個 tuple，包含一個 float 型態的變數和一個 Vec3 型態的變數。

在函式中，首先建立了三個 Vec3 型態的變數 v0、v1、v2，並將 vertices 中索引為 tri.p0、tri.p1、tri.p2 的元素分別賦值給這三個變數。接下來呼叫函式 rayPlaneIntersect，將 pos、dir、v0、v1、v2 五個參數傳遞給它，並將回傳值存入 t、p 變數中。接下來是一個條件判斷式，如果 t 等於 INF，就回傳 INF 和 None。

## model_loader類

**STL**

STL (STereoLithography) 檔案是一種用於 3D 打印的檔案格式，包含一個 3D 模型的平面三角形網格。可以使用多種軟體來讀取和編輯 STL 檔案，包括：

Blender: Blender 是一款免費的、開源的 3D 建模和動畫軟體。它可以用於讀取、編輯和儲存 STL 檔案。

SketchUp: SketchUp 是一款用於建築和工程設計的 3D 建模軟體。它可以讀取 STL 檔案，但無法儲存。

MeshLab: MeshLab 是一款免費的、開源的 3D 網格處理軟體。它可以讀取、編輯和儲存 STL 檔案。

可以使用這些軟體之一來打開 STL 檔案，然後你就可以看到模型並進行相應的操作（如旋轉、放大、縮小等）。

此外，還可以使用一些在網上提供的工具來在瀏覽器中查看 STL 檔案。例如，你可以使用 STL Viewer 工具在瀏覽器中載入和查看 STL 檔案。

**輸入範例**

```
// 加入三角面
add_triangle(Triangle(136, 135, 145))

// 加入頂點
add_vertex(Vec3(2.4117, 3.2277, 1.0585))
```

**Model Manage - 子類別**

```
class ModelManager:
    def init():
        Vec3 self.pos = Vec3(0, 0, 0)
        Matrix3 self.rot = Matrix3(Vec3(1, 0, 0), Vec3(0, 1, 0), Vec3(0, 0, 1))
        Vec3 self.scale = Vec3(1, 1, 1)
```

```
    endf

    def reset():
        n_vertices = 0
        n_triangles = 0
    endf

    def set_transform(Vec3 pos, Matrix3 rot, Vec3 scale):
        self.pos = pos
        self.rot = rot
        self.scale = scale
    endf

    def add_vertex(Vec3 v):
        vertices[n_vertices] = self.rot.transform(v * self.scale) + self.pos
        n_vertices = n_vertices + 1
    endf

    def add_triangle(Triangle t):
    endf

    def add_pyramid(Vec3 pos, float w, float h):
    endf

    def add_box(Vec3 pos, float w, float h):
    endf

    def add_spaceship():
    endf
endc
```

- init 函式用來初始化模型。它會將模型的座標（pos）、旋轉矩陣（rot）以及縮放比例（scale）設置為預設值。
- reset 函式用來清空模型中已存在的頂點和三角形的數量。
- set_transform 函式用來設置模型的座標、旋轉矩陣和縮放比例。
- add_vertex 函式用來向模型中添加一個頂點。它會將頂點應用模型的旋轉矩陣和縮放比例，然後將結果加上模型的座標，最後將轉換後的頂點儲存到頂點數組中。
- add_triangle 函式用來向模型中添加一個三角形。它的具體由外部STL模型輸入。
- add_pyramid 函式用來向模型中添加一個金字塔。它的具體由外部STL模型輸入。
- add_box 函式用來向模型中添加一個長方體。它的具體由外部STL模型輸入。
- add_spaceship 函式用來向模型中添加一艘飛船。它的具體由外部STL模型輸入。

## screen類

**build_char_level**

```
def build_char_level():
endf
```

函式會建立一個長度為 64 的陣列 charLevel，其中每個元素都對應到一個 ASCII 字符。它會將每個字符的 ASCII 碼和對應的索引值存到陣列中。例如，字符 "!" 的 ASCII 碼是 33，它就會被存到 charLevel[33] 中。

**push_render_buffer**

```
push_render_buffer Macro c, p
    mov eax, p
    mov ebx, c
    mov BYTE PTR buffer[eax], bl
    inc p
ENDM
```

是一個宏，它會將給定的字符加入到字符串 buffer 中，並將指標 p 加 1。

**display**

```
def display():
    int p = 0
    for i in range(HEIGHT):
        for j in range(WIDTH):
            int color = color_buffer[i][j]
            int char = charLevel[color]
            push_render_buffer char, p
            push_render_buffer char, p
        endl
        push_render_buffer 10, p
    endl
    push_render_buffer 0, p

    ; call Clrscr
    mov edx, OFFSET buffer
    call WriteString
endf
```

函式會將畫布上的內容轉換為字符串，並呼叫 WriteString 函式來顯示這個字符串。它會將畫布上每個像素的顏色值轉換為對應的 ASCII 字符，然後將這些字符加入到 buffer 字符串中。

**clear_buffer**

```
def clear_buffer():
    for i in range(HEIGHT):
        for j in range(WIDTH):
            color_buffer[i][j] = 0
            deep_buffer[i][j] = INF
        endl
    endl
endf
```

clear_buffer 函式會清空畫布的顏色和深度內容。它會將畫布的所有像素的顏色設為 0，深度設為無限大（INF）。

### test類

各種嘗試開發選項。

## 分工表

### 組長

- 黃守榕 60%
    - 演算法製作
    - 渲染器製作
    - 遊戲效率優化

### 組員

- 張勛皓 40%
    - 遊戲設計
    - 遊戲內容優化