Suprannua Engine Architecture Document

1. Introduction

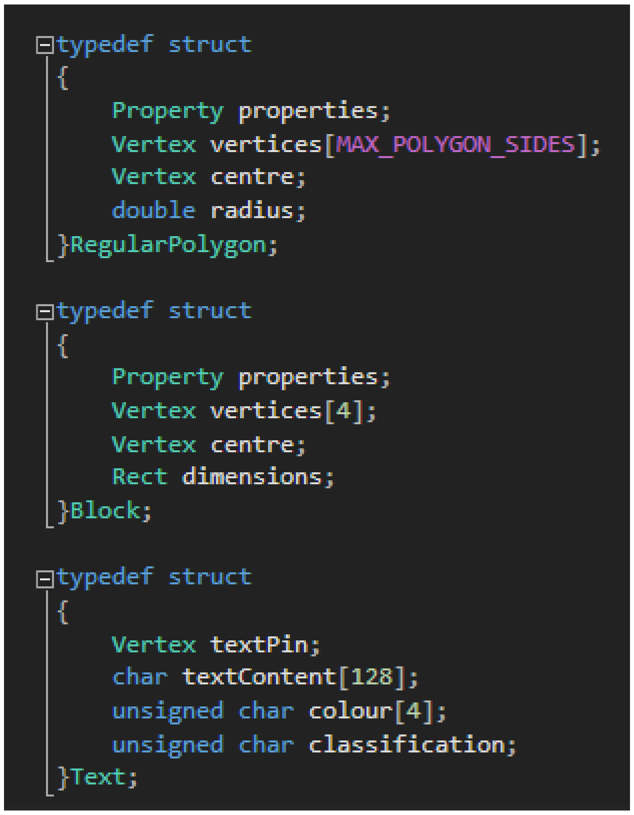
Suprannua Engine is a 2D platformer-oriented game framework for compiling simple, arcade-like PC games or visualisations for algorithms. The hallmark and namesake of this engine is a superannuated design where the visuals are minimally done with legacy OpenGL, and the architecture is structured as a collection of C functions.

This was a first time project that was designed beyond basic C programming exercises. Therefore, it was created with a limited knowledge of standard programming practices or even a solid plan of the architecture before having it implemented. It has since been refined to just get the components working as intended while being lenient on the coding style and original architecture (use of global variables, externs, etc).

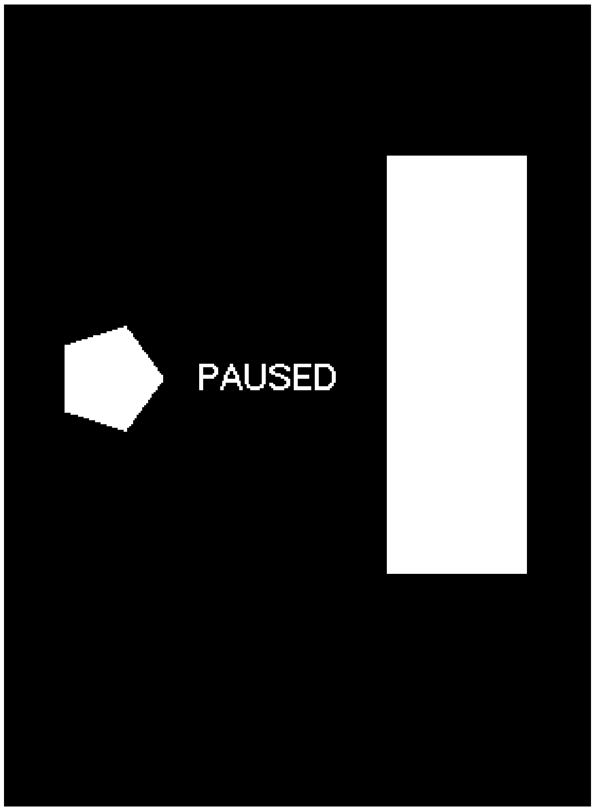
2. Core

2.1 Data structures

Suprannua Engine's game objects are strictly Polygons, Blocks, Texts and one Camera. All game objects are instances of type defined structs. They all contain the Vertex struct to represent x and y positions against a world map. The world map and rectangular objects like the Camera and Blocks all contain the Rect struct to represent width and height. The world map x coordinate, from 0 to the maximum size, corresponds with the left to right. The y coordinate from 0 to maximum size corresponds with down to up.



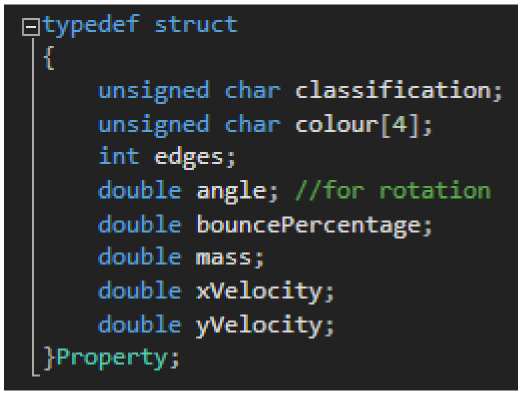
```
typedef struct
{
    Property properties;
    Vertex vertices[MAX_POLYGON_SIDES];
    Vertex centre;
    double radius;
}RegularPolygon;

typedef struct
{
    Property properties;
    Vertex vertices[4];
    Vertex centre;
    Rect dimensions;
}Block;

typedef struct
{
    Vertex textPin;
    char textContent[128];
    unsigned char colour[4];
    unsigned char classification;
}Text;
```



Polygons and Blocks are the only objects that contain the Properties struct, which describes the object's;

- •Classification (Whether it is a background, foreground, entity, platform, etc)
- •Colour (Including alpha)
- •Edges (Geometric sides)
- •Angle (Only really applicable to Polygons)
- •BouncePercentage (Elasticity)
- •Mass
- •xVelocity
- •yVelocity

Text is different where in addition to position, it stores a string with colour and either an entity or heads up display classification (to pin to the world map or camera, respectively).

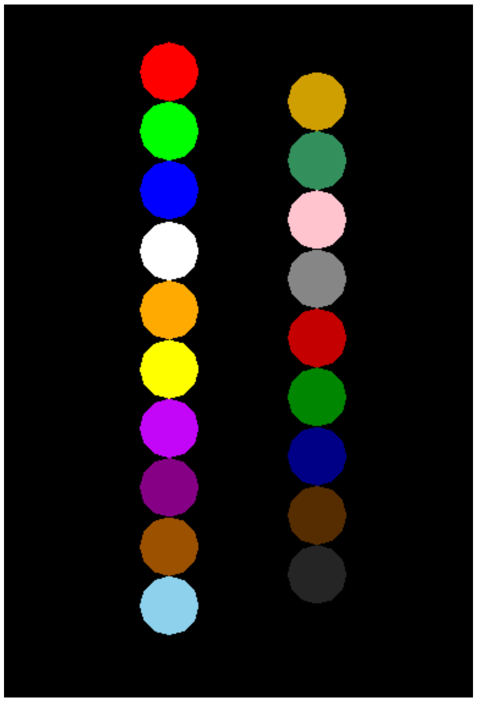


```
typedef struct
{
    unsigned char classification;
    unsigned char colour[4];
    int edges;
    double angle; //for rotation
    double bouncePercentage;
    double mass;
    double xVelocity;
    double yVelocity;
}Property;
```

2.2 Colour palette

There are 21 colours supported by the Suprannua Engine that range from black, white, rainbow colours and additional secondary colours with different shades. These values are stored in globally declared, hard coded arrays of unsigned chars:

```
const unsigned char black[3] = { 0,0,0 };
const unsigned char white[3] = { 255,255,255 };
const unsigned char red[3] = { 255,0,0 };
const unsigned char green[3] = { 0,255,0 };
const unsigned char blue[3] = { 0,0,255 };
const unsigned char orange[3] = { 255,165,0 };
const unsigned char yellow[3] = { 255,255,0 };
const unsigned char violet[3] = { 191,5,247 };
const unsigned char purple[3] = { 128,0,128 };
const unsigned char brown[3] = { 150,75,0 };
const unsigned char skyBlue[3] = { 135,206,235 };
const unsigned char gold[3] = { 204,153,0 };
const unsigned char seaGreen[3] = { 46,139,87 };
const unsigned char pink[3] = { 255,192,203 };
const unsigned char grey[3] = { 128,128,128 };
const unsigned char darkRed[3] = { 192,0,0 };
const unsigned char darkGreen[3] = { 0,128,0 };
const unsigned char darkBlue[3] = { 0,0,128 };
const unsigned char darkBrown[3] = { 80,40,0 };
const unsigned char magenta[3] = { 255,0,228 };
const unsigned char darkGrey[3] = { 32,32,32 };
```



The alpha values for the game objects are defined by the editor and text modules of the engine. By default the object is given a full 255 value so that they appear opaque. In the coming chapters, changing alpha values and transparency will be explained.

2.3 States

The engine holds input states in an array of boolean values called "keyStates." These booleans are indexed by ASCII values to register whether certain keys are pressed. For example, if the "a" key is pressed, the engine will take the ASCII value of "a", which is 97 and store a true value in array cell 97.

```
bool keyStates[128] = { false };
```

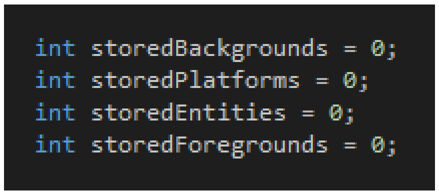
Having an array of these input states allow the engine to register multiple inputs at once so the player may move an object while performing another action at the same time, like running and jumping.

The gameState variable is primarily used for the game input definitions. It is an unsigned char that signals whether the game is considered to be in "GAMEPLAY", "MENU", or perhaps other custom states programmed by the developer.

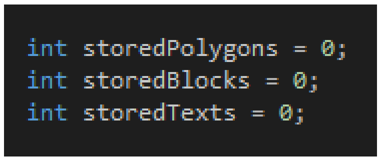
```
unsigned char gameState = GAMEPLAY;
```

For example, if the engine were in GAMEPLAY state, the WASD keys could be used for object movement, but if it were in MENU state, the WASD keys could be used for moving the camera or selecting and scrolling through a list of texts.

There are also variables for stored objects and renderables. Nothing is dynamically allocated, so these variables are used to keep track of the workload size of compute and rendering loop iterations.

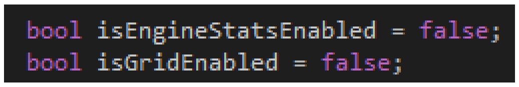


```
int storedBackgrounds = 0;
int storedPlatforms = 0;
int storedEntities = 0;
int storedForegrounds = 0;
```



```
int storedPolygons = 0;
int storedBlocks = 0;
int storedTexts = 0;
```

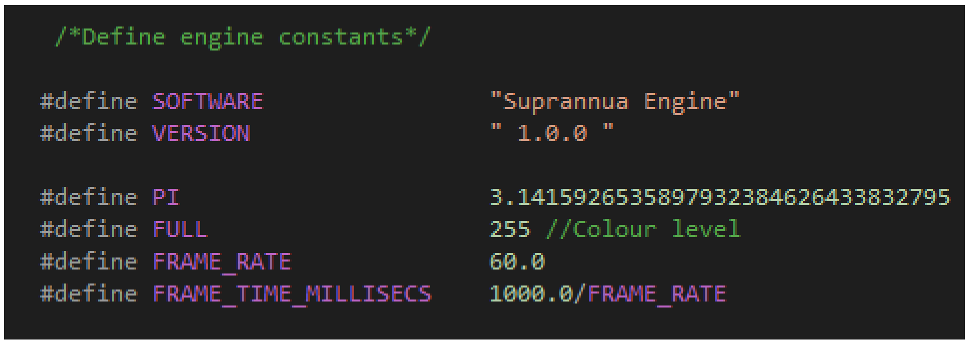
Finally, there are additional boolean variables for registering whether debug interfaces should be rendered to screen:



```
bool isEngineStatsEnabled = false;
bool isGridEnabled = false;
```

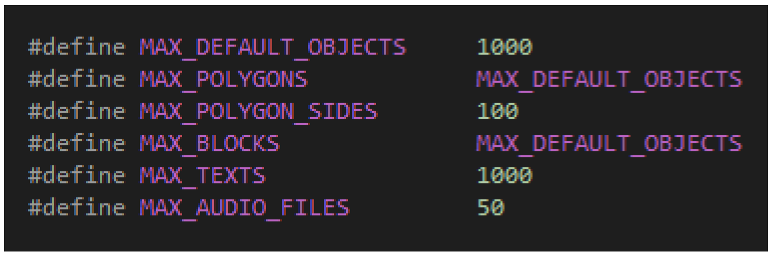
2.4 Constants

The engine retains constants for its name, and version number, and various other useful values like the value of pi to 32 places, max 8-bit colour value, target frame rate and the frame time in milliseconds.



```
 /*Define engine constants*/

#define SOFTWARE                "Suprannua Engine"
#define VERSION                 " 1.0.0 "

#define PI                      3.1415926535897932384626433832795
#define FULL                    255 //Colour level
#define FRAME_RATE              60.0
#define FRAME_TIME_MILLISECS    1000.0/FRAME_RATE
```

Other constants include the maximum amount of objects statically allocated for arrays, maximum polygon sides, texts and audio files to store.



```
#define MAX_DEFAULT_OBJECTS     1000
#define MAX_POLYGONS            MAX_DEFAULT_OBJECTS
#define MAX_POLYGON_SIDES       100
#define MAX_BLOCKS              MAX_DEFAULT_OBJECTS
#define MAX_TEXTS               1000
#define MAX_AUDIO_FILES         50
```

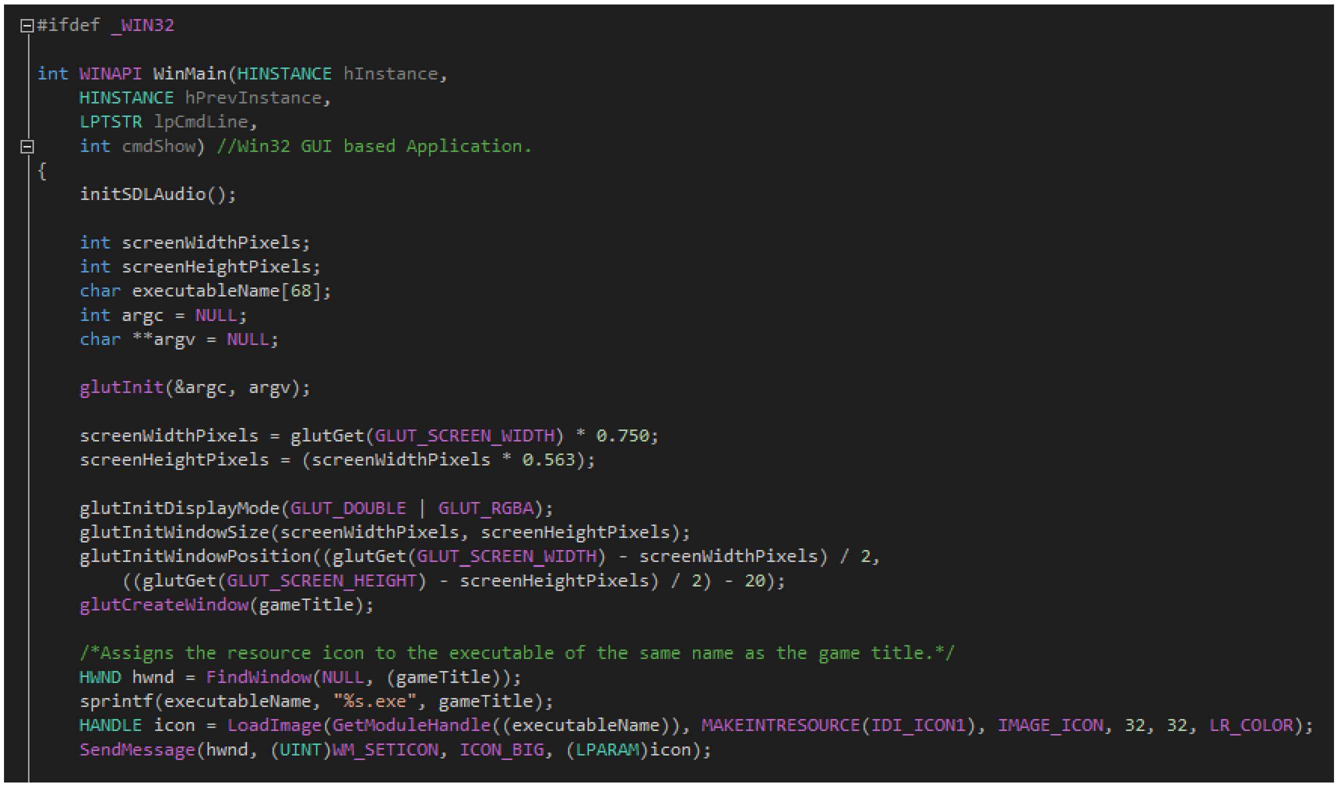
There are enumerations for the colour palette, objects, object types, object attributes, prepositions, control modes, spin directions, gravitation, platform scrolling directions and audio types to specify to major modules how operations on objects are to be performed.

2.5 Entry point

The Suprannua Engine at one point only used the WinAPI. This was to build the program with only a window and icon. Now by preprocessor directives, #ifdef _WIN32 compiles the WinAPI portion of code if it's running on Windows, and uses regular int main() on other platforms. Other platforms are supported provided that they have support for FreeGLUT and SDL.

3. FreeGLUT API

For the original API reference: http://freeglut.sourceforge.net/docs/api.php



```
#ifdef _WIN32

int WINAPI WinMain(HINSTANCE hInstance,
    HINSTANCE hPrevInstance,
    LPTSTR lpCmdLine,
    int cmdShow) //Win32 GUI based Application.
{
    initSDLAudio();

    int screenWidthPixels;
    int screenHeightPixels;
    char executableName[68];
    int argc = NULL;
    char **argv = NULL;

    glutInit(&argc, argv);

    screenWidthPixels = glutGet(GLUT_SCREEN_WIDTH) * 0.750;
    screenHeightPixels = (screenWidthPixels * 0.563);

    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGBA);
    glutInitWindowSize(screenWidthPixels, screenHeightPixels);
    glutInitWindowPosition((glutGet(GLUT_SCREEN_WIDTH) - screenWidthPixels) / 2,
        ((glutGet(GLUT_SCREEN_HEIGHT) - screenHeightPixels) / 2) - 20);
    glutCreateWindow(gameTitle);

    /*Assigns the resource icon to the executable of the same name as the game title.*/
    HWND hwnd = FindWindow(NULL, (gameTitle));
    sprintf(executableName, "%s.exe", gameTitle);
    HANDLE icon = LoadImage(GetModuleHandle((executableName)), MAKEINTRESOURCE(IDI_ICON1), IMAGE_ICON, 32, 32, LR_COLOR);
    SendMessage(hwnd, (UINT)WM_SETICON, ICON_BIG, (LPARAM)icon);
```

3.1 Window

As shown in the entry point section, the FreeGLUT API handles the OpenGL window context. The screen size is fixed at 75% of the monitor's width (retrieved by "glutGet(GLUT_SCREEN_WIDTH)") and in 16:9 aspect ratio. That ratio is enforced by multiplying the obtained screen width by (16/9 = 0.563).

"glutInitWindowPosition()" is given the centre of the screen with a 20 pixel downward shift as that appears to be the absolute height of a Windows titlebar. "gameTitle" is a globally defined value from the user defined code.

The window also automatically resists resizing by the "glutReshapeFunc()" which reiterates calculations to shape the window to be 75% of the screen width and at a 16:9 ratio.

3.2 Keyboard

From here, the keys are only being stored and not acted upon by a game. As mentioned in the "States (section 2.3)" the key value registered by the keyboard event is used to index an array of "keyStates" to map multiple truth values.

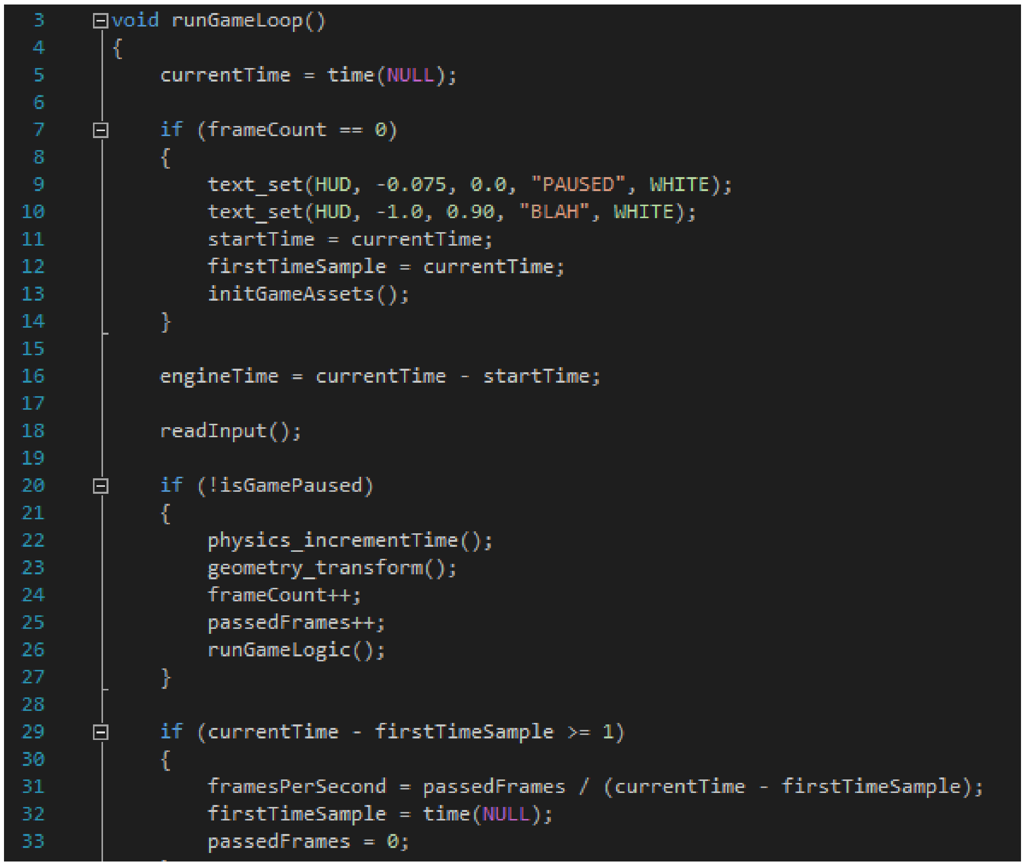
If the Escape key (ASCII value 27) is pressed, Suprannua Engine shuts down. Pressing the Tab key (ASCII value 9) centres the window if it is ever displaced during gameplay.

When the key is lifted in all cases, the "keyStates" are set back to false. When the "p" key is pressed and lifted, the "gameState" of changed between MENU and GAMEPLAY and the engine registers a pause or play. Also any audio playing in the background would be paused or unpaused simultaneously.

In the case that a soft reset is needed for softlocks or to return to the original state, pressing the "z", "c", and "r" keys simultaneously accomplishes this. This is an example of where the "keyStates" array is necessary. This method, however, does not reset the variables associated with the user defined code.

4. Game Loop

Suprannua Engine's game loop is simplistic. It is strictly frame based and it operates on universal time values for all modules. This makes every module operate with constant time slices during simulation. Because the engine is light on resources, under most circumstances this is negligible on gameplay. When a lag does occur, which is almost exclusive to a large body simulation of particles, it slows the runtime and preserves the frame by frame results.



```
void runGameLoop()
{
    currentTime = time(NULL);

    if (frameCount == 0)
    {
        text_set(HUD, -0.075, 0.0, "PAUSED", WHITE);
        text_set(HUD, -1.0, 0.90, "BLAH", WHITE);
        startTime = currentTime;
        firstTimeSample = currentTime;
        initGameAssets();
    }

    engineTime = currentTime - startTime;

    readInput();

    if (!isGamePaused)
    {
        physics_incrementTime();
        geometry_transform();
        frameCount++;
        passedFrames++;
        runGameLogic();
    }

    if (currentTime - firstTimeSample >= 1)
    {
        framesPerSecond = passedFrames / (currentTime - firstTimeSample);
        firstTimeSample = time(NULL);
        passedFrames = 0;
```

Every update within the engine's loop, including AI movements, is carried out as iterations of procedures that operate on the fixed and predictable time slices.

On the starting frame, the pause and engine stats text strings are initialized as well as the time values and the user defined game variables and procedures. The inputs are always read first on every iteration. Depending on whether the engine is paused or not, the update procedures for incrementing time, transforming geometry, and running game logic (calculations, scripts, everything) are called. Finally the frame rate calculations are done and optionally rendered along with an option grid layout. Besides these two renderables, backgrounds, platforms, entities, foreground and text are rendered, in that order.

5. Modules

5.1 2D Audio

For original API reference to SDL_mixer: http://sdl.beuc.net/sdl.wiki/SDL_mixer

The engine contains the SDL_mixer music and chunk data structures to store songs and sound effects respectively. These are statically defined within the 2DAudio translation unit. The audio section also contains variables that define the storage size of these structs.

The audio accepted by the engine are in the MP3 and OGG formats. These files are meant to have a 44.1kHz sample rate. At rates higher or lower than that, the audio may sound higher or lower pitched, respectively.

In the game initialization, all audio files are supposed to be registered with the "audio_set(type, filePath)" procedure with type representing whether the audio file will be a song or sound effect, and the filepath references the filename (including the extension).

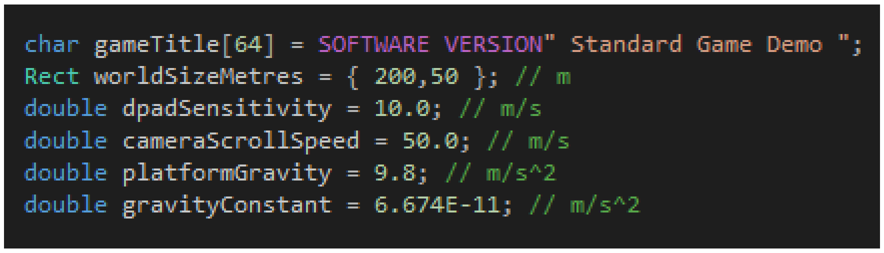
The "audio_play(type, audioNumber, loops)" procedure takes the audio number reference and plays it by the amount of times the loops specifies. The value "-1" for loops defines infinity, although the macro INFINITE is defined in the engine for this.

5.2 2D Camera
5.3 2D Renderer
5.4 AI
5.5 Editor
5.6 Events
5.7 Geometry
5.8 Input
5.9 Physics
5.10 Text

6. Game

6.1 Game global variables

There are variables that are defined by a custom game entry point file. These define the title, dpad sensitivity for camera and movement, world size in metres, gravity from platforms, and gravity between polygons.



```
char gameTitle[64] = SOFTWARE VERSION" Standard Game Demo ";
Rect worldSizeMetres = { 200,50 }; // m
double dpadSensitivity = 10.0; // m/s
double cameraScrollSpeed = 50.0; // m/s
double platformGravity = 9.8; // m/s^2
double gravityConstant = 6.674E-11; // m/s^2
```