SDL:Tutorials:Complete 2D Engine Graphics Core

From GPWiki

Contents

- 1 Graphics Core Overview
  - 1.1 How do I go about making a Graphics Core?
  - 1.2 How is the Graphics Core laid out?
- 2 Explanation of the Subsystems
  - 2.1 sTexture
  - 2.2 cTextureManager
  - 2.3 cAnimation
  - 2.4 cAnimationManager
  - 2.5 cFont
  - 2.6 cFontManager
  - 2.7 cCamera
  - 2.8 cGraphics
- 3 Example in action
- 4 Conclusion
  - 4.1 Final Notes
  - 4.2 About the Source
    - 4.2.1 Source and Runtime Demo

Graphics Core Overview

How do I go about making a Graphics Core?

Everyone knows what a graphics engine is suppose to do (draw pretty graphics) but how do you make one general core that supports everything you need? The first step to designing a something is to think about what you need. When I was designing the core I listed a set of things I wanted the graphics core to display, then narrowed everything down to a few simple groups. I came up with Textures, Sections of Texture, Animations, Fonts and a Camera. You may be wondering why I want a section of a texture, this is mostly to save resources.

How is the Graphics Core laid out?

My idea was to have a class for the type of graphics I wanted to use then have a manager class that handles creating of that graphic and also makes sure things only are loaded once. The manager class also handles clean up of the graphic, this resource style management makes it really easy to program for. So Here are the basic classes I have created and what they do:

- cAnimation - Holds information about one specific animation.
- cAnimationManager - Manages all animations.
- cCamera - Handles where the view is in the scene.
- cFont - Holds a font and all glyphs.
- cFontManager - Manages all fonts.
- sTexture - Hold information about a texture.
- cTextureManager - Manages all textures.
- cGraphics - Manages all manager systems.

As can be seen, this layout is straightforward and there is no real question concerning the functionality of each class. Now let's break down each class section-by-section to show you how the inner-workings of each of these subsystems were designed.

Explanation of the Subsystems

sTexture

The best subsystem to start out with would be the texture system. The texture system handles textures and texture sections, animations and fonts are built upon this system so its logical to start here. Here is the layout of my textures and texture sections.

```
struct sSection
{
    //u for width and v for height
    GLfloat uMin;
    GLfloat uMax;
    GLfloat vMin;
    GLfloat vMax;
};

struct sTexture
{
    GLuint Texture;
    std::string Filename;
    GLfloat Width;
    GLfloat Height;
};
```

```
struct sTextureSection
{
    uint TextureId;
    sSection Section;
    GLfloat Width;
    GLfloat Height;
};
```

A little explanation is needed for the way I layout my textures. First I have to say that everything that is drawn is done so in screen space, not in world space. this means that if you wanted to draw in the center of a 800x600 screen you would draw at 400,300 not 1.0f,1.0f like you would if you were using pure opengl. This is a convention I have chosen to make it easier on the user of the engine, screenspace has always made sense to me.
The sTexture structure holds an opengl texture, the filename that it loaded from and its width and height in pixels. The sTextureSection is similar, it holds an sTexture's id, an sSection, a width and a height. So now all that is left is the sSection structure. An sSection is just a set of points on a texture that define what portion of a texture you want to use. These points range from 0 to 1 for 'u' and 'v'. The reasoning for this is that you can replace the texture with a higher quality image that has the same proportions and it would cover the same area.
Ok so let me give an example. Lets say you want the upper left quarter of an image to be in a texture. Your coordinates would be (0,0) for the top, left and (0.5,0.5) for the bottom right of the section. Now we have to convert this to my section format, this is easy however. uMin and uMax represent the minimum and maximum coordinates in the horizontal direction. In our case uMin would be equal to 0 and uMax would be equal to 0.5 . Similarly vMin and vMax represent coordinates in the vertical direction. vMin would equal 0 and vMax is equal to 0.5 in this case. So if they are coordinate in the horizontal and vertical planes why not call them 'x' and 'y' like normal people? The reasoning behind calling the coordinates 'u' and 'v' is because they are not actual coordinates on the screen rather they are relative to a texture. In otherwords 'u' and 'v' are used just to distinguish what they are used for.

cTextureManager

The texture manager handles loading and drawing of a texture. The texture manager also handles more advanced techniques such as texture tinting, scaling and rotations. Here is what the class looks like:

```
class cTextureManager
{
    public:
        cTextureManager();
        ~cTextureManager();

        uint LoadTextureFromFile(std::string filename);
        uint LoadTextureFromMemory(SDL_Surface* Surface, std::string filename = "");

        void ReloadTextures();

        uint CreateTextureSection(uint TextureId, sSection Section);

        void RemoveTexture(uint TextureId);
        void RemoveTextureSection(uint SectionId);
        void DrawTextureSection(uint SectionId, GLfloat X, GLfloat Y, GLfloat Scale = 1,
            GLfloat Rotatation = 0, GLfloat red = 1, GLfloat green = 1, GLfloat blue = 1);
        void DrawTexture(uint TextureId, GLfloat X, GLfloat Y, GLfloat Scale = 1,
            GLfloat Rotatation = 0, GLfloat red = 1, GLfloat green = 1, GLfloat blue = 1);

        GLfloat GetTextureWidth(uint Texture);
        GLfloat GetTextureHeight(uint Texture);

        void RemoveAllTextures();
        void RemoveAllTextureSections();

    protected:
        std::map< uint, sTexture > m_Textures;
        std::map< uint, sTextureSection > m_TextureSections;
};
```

Most of the class should be easy to follow but I'll clarify what their uses are. 'LoadTextureFromFile' Does what you think it would, it loads a texture from a file. 'LoadTextureFromMemory' is useful when you already have a texture loaded from another resource such a custom resource file, in my examples I do not use this but it will be useful later on. 'ReloadTextures' is used whenever you lose all the textures in the screen. Textures must be reloaded when the window is resized or put into fullscreen.
'CreateTextureSection' makes a section of a texture given a texture id and the section you wish to use. 'RemoveTexture' and 'RemoveTextureSection' remove that item from memory. Keep in mind if you remove a texture then all texture sections that used that texture will be deleted as well.
Now comes the fun part, drawing. Since both 'DrawTextureSection' and 'DrawTexture' have the same interface I will only explain one of them. 'DrawTexture' takes a texture id, an 'x' and 'y' coordinate and draws it to the screen. Optional you can chose to scale the image, rotate it or tint the texture. For reference I made the rotation the angle in degrees.
If you wish to know how big a texture's size is then you can use the 'GetTextureWidth' and 'GetTextureHeight' functions. The 'RemoveAllTextures' and 'RemoveAllTextureSections' do exactly what you think.
Lastly we have two private variables. 'm_Textures' which holds all textures and 'm_TextureSections' that holds all texture sections. If you do not know what map does I recommend that you look it up, in short it is a way to "map" one thing to another. In my case I have mapped unsigned integers to my texture structures.
If you haven't noticed I use id's for my textures instead of pointers. This makes it safer for the user so they can't mess with the internals of texture. Upon creation of a texture you get an id, you must save this id for drawing or whatever function you wish to use it in later on. Keep in mind that if you get a 0 returned as an id this means that the item was not created.

cAnimation

Now that we have textures done we can expand upon it with animations. Here is my class I have for animations:

```
class cAnimation
{
    private:
        std::vector< uint > m_Textures;
        std::vector< Uint32 > m_Delays;
        uint m_CurrentFrame;

        Uint32 m_LastTime;

        bool m_Paused;
        int m_Repeations;

    public:
        cAnimation();
        ~cAnimation();

        void Update();
        void Pause();
        void Resume();
        void Reset();
        void JumpToFrame(uint Frame);

        void SetRepeations(int Repeations);
        void SetTexture(uint Texture, uint position);
        void SetDelay(Uint32 delay, uint position);
        void Draw(GLfloat x, GLfloat y, GLfloat Scale = 1, GLfloat rotation= 0,
            GLfloat red = 1, GLfloat green = 1, GLfloat blue = 1);

        uint GetFrameCount();
};
```

As you can see the animation class has a lot of functionality. So let me explain what everything is for. As you've probably guessed 'm_Textures' holds all the textures in the animation. 'm_Delays' holds the individual delay for each frame. 'm_CurrentFrame' contains the position in the animation we are in. 'm_LastTime' holds the last time the animation has been updated. 'm_Paused' tells if the animation is currently paused or not. 'm_Repeations' holds the number of times to repeat the animation, -1 tells the animation to repeat infinitely.
Now that I have explained the internal variables of the animation class I will explain what the functions do. 'Update' will see if enough time has passed and change the animation accordingly. 'Pause' will pause the animation. 'Resume' unpauses the animation. 'Reset' will reset the animation to the first frame. 'JumpToFrame' will set the animation to whatever frame you chose.
'SetRepeations' will change the number of times an animation will repeat. 'SetTexture' will change the texture the at the given frame. If the frame is bigger than the animation then it will append it to the end of the animation. 'SetDelay' will change the delay for the frame given, unlike setting a texture it only works for a valid frame. 'Draw' makes a call to the texture manager to draw the current texture. Lastly 'GetFrameCount' will return how many frames are in the animation.

cAnimationManager

Now that we have defined animations we can create an animation manager. The style of the animation manager is much the same as the texture manager.

```
class cAnimationManager
{
    public:
        cAnimationManager();
        ~cAnimationManager();

        uint CreateAnimation( std::vector< uint > Frames, std::vector< Uint32 > Delays);
        uint CreateAnimation( std::vector< uint > Frames, Uint32 Delay);

        void Remove(uint Animation);
        void RemoveAll();

        void Pause(uint Animation);
        void PauseAll();

        void Update(uint Animation);
        void UpdateAll();

        void Resume(uint Animation);
        void ResumeAll();

        void SetRepeations(uint Animation, int reps);

        void Reset(uint Animation);
        void ResetAll();

        void JumpToFrame(uint Animation, uint Frame);

        void DrawAnimation(uint Animation, GLfloat x, GLfloat y, GLfloat Scale = 1,
           GLfloat rotation = 0, GLfloat red = 1, GLfloat green = 1, GLfloat blue = 1);

    private:
        std::map< uint, cAnimation > m_Animations;
};
```

Like the texture manager the animation manager also returns an id for the animations created. The only internal variable is 'm_Animations' which holds all the animations in the system. The functions in the animation manager mirror that of the animation class so I won't go over them all again. I will point out that there are two ways to call each function in an animation. The first way is to a single call to an animation via its id. The second way is to apply the function to all the animations. The most useful function in my opinion is 'UpdateAll', it goes through all animations and updates their frames (nifty huh?). The only real difference from the texture manager is in creation of an animation. There are two ways to create an animation the first way uses a vector of delays and the second way has only one delay that is applied to each frame. The 'Frames' variable is the same for each way, it is a vector of texture ids.

cFont

Before I show you how I've designed the font structures I will tell you how I decided to implement my fonts. I must admit I went the easy way out and decided to use textured fonts, at first I was a little weary of doing it this way but the results turned out better than expected.
A font has two files, the first one being an image with all the letter and the second one is a text file that contains all the glyphs and their coordinates on the image. Take a look under "Data/Fonts/" if you wish to see exactly how it works.
Now that I've said that I'm using textured fonts, here are the structures I use:

```
struct sGlyph
{
    char Glyph;
    uint TextureSection;
};
```

```
struct sFont
{
    GLfloat Spacing;
    GLfloat GlyphWidth;
    GLfloat GlyphHeight;
    std::string Name;
    uint Texture;
    std::map<char, sGlyph> Gylphs;
};
```

First let me explain what a glyph is for those that don't know. Basically a glyph is any letter or symbol. A symbol can be anything but I'm sticking to the basics such as braces, brackets, periods and all those fun things we have come to love in English (and other languages). Now I can explain the glyph structure. The structure is really quite simple, it has two variables, the glyph it represents and the texture section that contains that glyph. The sFont structure needs little explanation. The 'Spacing' variable tells how far apart we space between glyphs. 'GlyphWidth' and 'GlyphHeight' are the sizes of the glyphs in the font. 'Name' is the name of the font. 'Texture' is the id of the whole font texture. And 'Glyphs' contains all the Glyphs in the font.

cFontManager

With every new structure comes a manager, the font system isn't any different.

```
class cFontManager
{
    public:
        cFontManager();
        ~cFontManager();

        void SetFontPath(std::string Path);
        void SetLetterSpacing(std::string Font, GLfloat Spacing);

        std::string LoadFont(std::string FontName);
        void ReleaseFont(std::string Font);
        void ReleaseAllFonts();

        void DrawText(std::string Font, std::string Text, GLfloat X, GLfloat Y, GLfloat Scale = 1,
            GLfloat Rotation = 0, GLfloat red = 1, GLfloat green = 1, GLfloat blue = 1);

    private:
        std::string m_FontPath;
        std::map<std::string, sFont> m_Fonts;
};
```

As you can see there really isn't much to the font manager. Before you can load any font you need to specify the path to the fonts, you can do with 'SetFontPath'. 'SetLetterSpacing' as you've probably guessed sets the spacing between glyphs for the font. The 'LoadFont' works a little different than the texture and animation managers as it returns a string instead of uint as the id. 'ReleaseFont' and 'ReleaseAllFonts' remove fonts from the system. The real workhorse is in 'DrawText', it takes in a font and a line of text plus all the standard drawing enchantments I've talked about. There are two internal variables. 'm_FontPath' holds the path to all the fonts that will be loaded. 'm_Fonts' holds all the fonts in the system.

cCamera

Although all the drawing functions have been presented there is still some more things the graphics core needs to be complete. The camera is a very useful class as it allows you to scroll around the screen and look at different things. Here is how my camera class is laid out.

```
class cCamera
{
    public:
        cCamera();
        ~cCamera();

        void Move(GLfloat x, GLfloat y);
        void SetPosition(GLfloat x, GLfloat y);

        GLfloat GetXposition();
        GLfloat GetYposition();
    private:
        GLfloat m_Xpos;
        GLfloat m_Ypos;
};
```

As you can see the camera class is really simple, up can move it by a relative value or you can set an absolute position. When a texture goes to draw it first gets the values from the camera and translates to the position, it is as simple as that.

cGraphics

The last thing I decided to make is a manager to rule them all. This just saves me some time when initializing the graphics engine. Here is the class to manage all subsystems.

```
class cGraphics
{
    public:
        cGraphics();
        ~cGraphics();

        bool Initialize(int Width, int Height);
        void Shutdown();

        uint CurrentTexture;

    protected:
        bool m_Loaded;
};
```

Basically what happens is that when 'cGraphics' is created all of the other managers are created and initialized. 'Initialize' sets up the window with an orthogonal projection with the given width and height. 'Shutdown' kills all the managers and all their resources. 'CurrentTexture' holds the current the texture id that in graphics memory. Lastly 'm_Loaded' tells if the graphics core is loaded or not. The reason this is useful is because sometimes you wish to reinitialize the screen when you resize the screen or switch into fullscreen mode. If you were to call 'Initialize' after it is already loaded it will reload all textures and remake the screen, this is quite useful.

Example in action

Here is the code I use in my demo program. It's fairly easy to follow.

```
//intialize the graphics engine
g_Graphics->Initialize(m_Width, m_Height);

//load the initial texture
uint texture = g_TextureManager->LoadTextureFromFile("test/picture.png");


//make four sections of the image, one for each quadrant.
sSection section1;
section1.uMax = 0.5f;
section1.uMin = 0.0f;
section1.vMax = 0.5f;
section1.vMin = 0.0f;

sSection section2;
section2.uMax = 1.0f;
section2.uMin = 0.5f;
section2.vMax = 0.5f;
section2.vMin = 0.0f;

sSection section3;
section3.uMax = 0.5f;
section3.uMin = 0.0f;
section3.vMax = 1.0f;
section3.vMin = 0.5f;

sSection section4;
section4.uMax = 1.0f;
section4.uMin = 0.5f;
section4.vMax = 1.0f;
section4.vMin = 0.5f;

//using the sections make the texture sections
uint textureSect1 = g_TextureManager->CreateTextureSection(texture, section1);
uint textureSect2 = g_TextureManager->CreateTextureSection(texture, section2);
uint textureSect3 = g_TextureManager->CreateTextureSection(texture, section3);
uint textureSect4 = g_TextureManager->CreateTextureSection(texture, section4);

//make the frame of an animation from the sections
vector< uint > Frames;
Frames.push_back(textureSect2);
Frames.push_back(textureSect1);
Frames.push_back(textureSect3);
Frames.push_back(textureSect4);

//make two animations from the same frames but use different delays
uint anim1 = g_AnimationManager->CreateAnimation( Frames, 1000 );
uint anim2 = g_AnimationManager->CreateAnimation( Frames, 500 );


//load a font
g_FontManager->SetFontPath("Data/Fonts/");
string font = g_FontManager->LoadFont("franks");
g_FontManager->SetLetterSpacing(font, 0.55f);


//setup some variables to handle event process (this will be moved later on)
SDL_Event event;
bool quit = false;
```

```
//these variables are for the objects that have scaling and rotation applied to them
GLfloat Scale = 1.0;
bool shrink = true;
GLfloat Rotation = 0;

// these variables are used to calculate the fps
Uint32 LastTime = SDL_GetTicks();
uint ShownFrames = 0;
ostringstream FPS;

//draw 20 frames
while(!quit)
{

    // more event processing is here (again this will be moved out later on)
    while ( SDL_PollEvent( &event ) )
    {
        switch( event.type )
        {
            case SDL_VIDEORESIZE:
                ResizeWindow( event.resize.w, event.resize.h );
                break;
            case SDL_QUIT:
                /* handle quit requests */
                quit = true;
                break;
            default:
                break;
        }
    }

    //calculate the fps
    ShownFrames++;

    if((SDL_GetTicks() - LastTime) >= 1000)
    {
        FPS.str("");
        FPS << ShownFrames;

        ShownFrames = 0;
        LastTime = SDL_GetTicks();
    }



    //update the rotation angle
    Rotation += 1;

    if(Rotation == 360)
    {
        Rotation = 0;
    }

    //update the scaling factor
    if(Scale <= 0.5f)
    {
        shrink = false;
    }
    if(Scale >= 1.5f)
    {
        shrink = true;
    }

    if(shrink)
    {
        Scale -= 0.02f;
    }
    else
    {
        Scale += 0.02f;
    }

    //clear the screen (debating on moving this into cGraphics)
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    //move the camera a little bit each frame
    g_Camera->Move(0.5f, 0.5f);

    //update and draw all animation
    g_AnimationManager->UpdateAll();
    g_AnimationManager->DrawAnimation(anim1, 20, 20);
    g_AnimationManager->DrawAnimation(anim2, 200, 200, Scale, Rotation, 1, 0, 0);

    //draw a few text objects on the screen
    //(NOTE: subtracting the camera position from drawing makes draw absolute on the screen, ignoring the
camera)
    g_FontManager->DrawText(font, "Testing", 300 - g_Camera->GetXposition(), 300);
    g_FontManager->DrawText(font, "The Quick Brown Fox Jumps Over The Lazy Dogs Back",
        -1000, 250 - g_Camera->GetYposition(), 2.0f, 0, 0, 1, 0);
    g_FontManager->DrawText(font, "FPS: " + FPS.str(), 10 - g_Camera->GetXposition(),
         560 - g_Camera->GetYposition(), 0.5, 0, 0, 1);

    // flip the screen and delay a little to save cpu
    SDL_GL_SwapBuffers();
    SDL_Delay(1);
}
```

I went through and commented the example code as much as possible so it is even easier to follow. As you can see you can do a lot with very little code, best of all it all looks clean and no pointers :).

Conclusion

Final Notes

As you can see designing a graphics core isn't that hard. It is much easier to design an engine when you know what you want in your engine. The engine I have designed isn't as full featured as my final version will be but its a good start and is easily extensible.

About the Source

Last my code was in xcode, since then I have converted it over code::blocks (I told you it was portable right). The libraries that are need are OpenGL, SDL and SDL_image.

Source and Runtime Demo

Kore Source and Runtime Demo

This tutorial was written by Seoushi. Do you like this tutorial? Have any questions or comments? Let me know or you can ask the forums.

Retrieved from "http://gpwiki.org/index.php/SDL:Tutorials:Complete_2D_Engine_Graphics_Core"

Categories: C and SDL | All C articles | All SDL articles