Handmade lite

Technical Reference

**Copyright Notice**

Please note that this project is not open source and therefore has no license affiliated with it. However, there really is nothing stopping you from using the engine to build your own game or learn from the code by viewing it and copying the odd function or class. Feel free to use Handmade as an educational tool. Just don't blatantly make a direct copy of everything and claim it as your own piece of work. While you may get away with it being on the opposite side of the planet, it really isn't very nice to steal someone else's hard work and make it your own. Enough people are doing that already - rather remain an honest person.

**Table of Contents**

**Introduction**

…

**Class Design**

The following section will describe how the classes and code in the game engine is laid out, with detailed descriptions of the data members, member functions and what they aim to achieve. Furthermore, the reasoning behind why certain choices were made when designing the code will be highlighted here as well.

**Audio**

This class encapsulates a music object that is able to play any background song or a voice over piece. The volume of the audio can be set with any value between 0 and 128. The main member variable is the music pointer, which can be set by requesting the object from the Audio Manager and stating whether you require a music or voice object. When playing the audio, the loop counter can be passed in to tell the audio system whether to play the audio only once or loop it endlessly.

**SFX**

This class encapsulates a simple sound or sound effect, for example a gun shot, smashing glass, cracking wood, etc. The volume of the sound effect can be set with any value between 0 and 128. The main member variable is the sound effect pointer, which can be set by requesting the object from the Audio Manager. When playing the sound effect, the loop counter can be passed in to tell the audio system how many times to loop the sound. By default, the sound will only be played once.

**AABB**

This class encapsulates an axis-aligned bounding box that determines collisions that may occur with other bounding boxes. It will calculate the min and max values of the bounding box which are then used to calculate for collision, using a particular formula. For the min and max values to be correctly calculated, the position and dimension of the box need to be set and for that there is a SetPosition() and SetDimension() routine, respectively. Each one will internally call the CreateBox() function that will calculate the min and max values. This is so that each time the position or dimension change, the bounding box volume is correctly updated. The IsColliding() function does all of the work and uses the min and max X and Y values within a formula to calculate for collision. Note that the min and max calculations are flipped on the y axis compared to the original OpenGL Handmade Game Engine, because the y axis is flipped in SDL.

There is a need for a separate private CreateBox() routine so that the min and max values can be determined. If the min and max values are calculated only in the IsColliding() routine, then the second box's min and max values will never be known, as you would have to call both IsColliding() functions at once - which is very impractical. The Sphere class on the other hand only needs the position and radius so no separate CreateSphere() routine is needed.

**Sphere**

This class encapsulates a sphere colliding volume that determines collisions that may occur between two spheres. It will make use of the position and radius of either sphere and use a particular formula in the IsColliding() routine to check if the two spheres collide. Because the objects in SDL all have a position at the top left corner of the sprite or image, the sphere's collision volume needs to perform an extra calculation to use the centre position of each sphere instead. This is all handled within the IsColliding() routine.

**Sprite**

This class encapsulates sprite objects and uses a texture pointer to represent the actual sprite image. The image can be a single image, or a spritesheet of multiple images, which is great for animations. The main aim of this class is to be able to assign single images or a specific spritesheet cell image to be used. The Sprite class is a base class from which the Animation class derives from.

The enumerated type is there to set the sprite orientation. Either the sprite will be rendered as is, or it can be flipped on the horizontal or vertical axis. This value can be set directly in the Draw() function when the sprite gets rendered.

One of the most important member variables is the texture index variable. It is the index value of the actual sprite cell of a sprite sheet, if the texture image has multiple images within. If the texture image is singular, the texture index is 0 by default. The texture index is zero based and is a whole number value counting from top left and going right and down the spritesheet.

It is calculated differently in the base class to the Animation sub-class.

The setter functions will set the specific properties of the sprite. The SetTexture() routine is used to assign an SDL texture object to the sprite object. This pointer is requested from the Texture Manager. The SetTextureDimension() will set the dimension of the texture in terms of rows

and colums. A single texture will be 1x1 but a spritesheet might have multiple columns and rows. The function also takes in the dimension of each texture cell and if its a singular texture, the dimension will be the total dimension of the texture. This has been dramatically changed in comparison to the original Handmade engine code, because in OpenGL we work with normalized values

and here we deal with full pixel values and sizes. The SetSpriteDimension() routine is there to assign the size of the sprite as it should appear on screen.

The Draw() function is different in all the classes, therefore it will be virtually overriden based on what object type is being used. The base class and the sub-class all calculate the texture index differently. The function will call the internal BlitSprite() routine which takes thesprite and screen render destination dimensions and calculates how to "cut out" the sprite and how to display it accordingly on screen.

**Animation**

This class encapsulates animation objects and derives from the Sprite base class. Animations generally use spritesheets with multiple images and loop through them either endlessly or for one cycle only. This setting can be set externally from the client code. The animation speed can also be set externally.

The IsAnimationLooping() function returns a bool reference variable, which makes this routine a getter/setter. This makes setting the flag in the client code a lot easier. The IsAnimationDead()function can be called from the client code to determine if an animation is complete so that the game object that contains it may be destroyed. This is needed so that the game object is only destroyed once its animation components has finished rendering. For example an asteroid game object first needs to render its explosion animation before its removed from memory.

Because the texture index value is calculated differently here than in the Sprite base class, the Draw() function is overriden. There are a few other checks inside the Draw() routine that check whether the animation loops or not. Sometimes an animation will loop endlessly (like a player's walk cycle) or just loop once (like an explosion). When the animation is set to loop only once, the first texture cell in the animation has to come around again before the animation is finally

set to end. This is because if we were to end it at the last cell, the last cell wouldn't be drawn so instead we let the entire spritesheet draw before killing the animation.

- overloaded functions make great ways to create getter/setter combos!

**Text**

This class encapsulates text objects that are able to store and display regular text strings on screen using a TTF font pointer that references a particular font style and size. It uses the font style, a particular text color and a string of text to create a texture that will be drawn on screen, almost like a sprite. However, this class does not derive from the Sprite class, unlike the OpenGL version of Handmade, as it shares very little functionality with the Sprite class.

There are a few setter functions to set the various properties of the text object. The SetFont() routine is used to assign a TTF font object to the text object. This pointer is requested from the Texture Manager. Whenever the color and text are set, via SetColor() and SetText(), the internal CreateText() function is called each time, so that the text object can be re-created before it's drawn in the Draw() routine. There is no need to re-create the text each frame, as this becomes expensive considering the work needed to create the text object.

The CreateText() function will create a text object and store it as a regular SDL surface before it's converted into a SDL texture. This texture is then used to render on screen, just like a sprite renders its images on screen. Because there is a lot of loading and unloading SDL objects in this routine, it becomes expensive to call it each frame, so its only called when needed, ie when the color or text string of the text object changes.

**GameObject**

This class encapsulates a game object that will exist in the game world and represent any type of object like a player, enemy, weapon, inventory item, etc. It is designed around the same principle as Unity's game objects and will be instantiated in the client code at runtime from within the game specific code. It consists of three main flags, m\_isAlive, m\_isActive and m\_isVisible, to signify if the object is exisiting in memory, being used in the game world and hidden or not from the game world, respectively. Each game object also has a string tag to

identify it amongst other objects, and a priority value for 2D game worlds without a Z-axis, so that game objects may be drawn in a specific order to give the illusion of depth. All game objects need to be instantiated as sub-classes of this abstract base class.

The getters and setters are split into normal ones and combo variants. The combo variants, IsAlive(), IsActive() and IsVisible() return references to the respective variables because these flag variables may be used frequently and getting and setting them individually may become inconvenient.

Each game object has a Update() and Draw() function which will be overriden in the sub-class, where the actual main functionality of that object will exist. Generally the Update() routine is there for the updating of all position, rotation and scale values, alongside others. The Draw() function is intended to render all internal components and other objects.

**Background**

This class is designed to represent the background image and music for each game state. Therefore, it takes in image and audio filename arguments to correctly load and use the right resources. Ideally, this class should be instantiated within each game state as there will be different backdrops and music playing in each one.

**MainMenu**

This class encapsulates the basic default main menu that is presented to the player in both the Menu and End states. The menu can be altered to display any menu choice the user desires and once the game is running, will react to key presses to manoeuvre through the menu. Using SetMenuText(), the menu options can be set accordingly and GetMenuOption() returns the index value of the menu choice, once the user presses the ENTER key whilst in-game. Note that

all menu item values are zero-based! The menu can also be refreshed using Reset() so that new menu choices can be accepted.

MenuKey is more automated. The Assets/Fonts sub-directory is already set in here so that on the higher level, you just add in the font name. If you ever change the directory, you must change it here, however by default this location is best suited so no need to change it!

The m\_posX and m\_posY variables are there to store the 'origin' positions from which the x and y position of each text object will be based

**GameState**

This class encapsulates a game state that will exist within the game. A game state can be anything like a play state, pause state, menu state, etc and will contain all the game specific code specific to that state. Every game state needs to be instantiated as a sub-class of this abstract base class.

The main flag variable m\_isActive will keep track if the game state is currently active or not. As long as its active, its overriden Update() and Draw() routines will constantly be called until the state is set as not active, at which point the state will close down.

The OnEnter() routine is called when the state is activated and this function will load all resources from file, instantiate the game objects for that state and do all start up tasks for the active state. The opposing OnExit() function will do the opposite when the state is deactivated, ie - call all shutdown tasks, remove all game objects from memory and free all resources from memory for the given state.

**MenuState**

This game state monitors the beginning state of the game. Things like menu's, start up screens, splash screens, etc are intended to be used here. This particular state consists of a background object, which represents the backdrop image and audio played, and a menu to offer the player some start-up choices. For now the default menu choices are to play or quit the game.

**PlayState**

This game state controls the main state of the game. It consists of a background object which will render the main bacdrop image and play the correct background music. Other game objects can be added to the vector as needed.

The Update() function will update all the game objects in the vector, as long as they are active. The Draw() routine draws all the game objects in the vector, as long as they are active and visible.

**EndState**

This game state monitors the final state of the game. Things like credits, end screens, final messages, etc are intended to be used here. This particular state consists of a background object, which represents the backdrop image and audio played, and a menu to offer the player some ending choices. For now the default menu choices are to play again or quit the game for good.

**Singleton**

This is a Singleton template class. It will instantiate another class of any type and allow access to it via the Instance() member function.

The constructor, copy constructor and assignment operator all need to be private so that this class cannot be individually instantiated.

**Main**

This is the main starting point for the game demo. The main game state is created here and added to the main game manager, from where it is run. Change the screen width and height accordingly, and feel free to add more game states.

**Game**

This class controls the main game. It is a Singleton so that it can be accessed from anywhere in the main code, and because only one game can exist at any one time. At its core, it consists of a deque of game states, that can be added and removed at any time from within the main client code. The main game loop that controls the entire game and its game states runs from within the Run() function.

The AddState() routine will add a temporary game state to the FRONT of the deque. This is ideal for states such as a pause or instruction state. The previous state remains in the deque and as soon as the temporary state is complete, it is removed again. The ChangeState() function will add a state to the BACK of the deque and remove the previous state from the FRONT. This is ideal for when game states transition from one to another.

The active state will have its OnEnter(), Update(), Draw() and OnExit() routines called accordingly from within this class. As soon as the deque contains no more states, the game will end.

This class also keeps track of all time passed. It records elapsed time which is great for frame-independent games, where physics or graphics updates rely on time instead of frame rate. The class also keeps track of total time passed since the game first started. The Run() function calculates the elapsed time for each frame.

The Initialize() and ShutDown() functions start up and close down all the other managers respectively.

**AudioManager**

This class encapsulates an audio manager which will load music, sound effects and voice audio files from disk and store them as SDL sound objects in a map, with a string reference to that audio object. Currently the audio files supported are MP3, WAV, OGG, FLAC, <TBA>. To access any particular audio inside the map the string reference is needed to sift through the map and find the audio. Audio objects can also be unloaded from memory individually or in bulk. This class is

a Singleton.

Two enum types have been created. The RemoveType enum is used when removing audio objects from the map. Either one single specific SDL audio object can be removed, or the entire map of audio objects can be cleared. The AudioType enum is there for letting the manager class know what type of audio it will dealing with.

To use this class, an Audio object needs to be created inside the game client code. Then that audio object will be able to link to a specific SDL audio object from within the Audio Manager. The GetSFXData(), GetMusicData() and GetVoiceData() functions are used to request the SDL sound pointer and link the audio data with the audio object.

The LoadFromFile() routine will load in the audio data from a file and store the audio in the correct audio map. Both the music and voice audio is loaded into memory in a streamed way, so as to save on memory because these files can become plentiful and large. SFX files are loaded in directly into memory and kept there. Streamed audio is CPU intensive!

An Output() routine is there for debug purposes only and will print to the console how many audio objects are currently stored in each of the three supported maps.

- better to have a handle on the audio data. We could have just used static Play(), etc but then

we are digging in the map every time!

**InputManager**

This is a Singleton Manager class that controls all the input from the mouse and keyboard (More controls later!!) It uses a set of SDL library tools to manage the input and stores the mouse and keyboard values in various property variables. The class as a whole can determine what keys have been pressed and what state the mouse buttons are in, and these states can be queried from external code at any time.

Three enumerated types have been created to be used when storing the button states of the mouse buttons, the mouse cursor state, and the SDL mouse cursor flag values, whose values all correspond with SDL's numeric values for those specific mouse cursors.

The class variables store all the details of the mouse and keyboard and the m\_isXClicked and m\_isKeyPressed variables store whether the X in the top right corner of the game window has been clicked or whether a keyboard key has been pressed or not respectively. The m\_isKeyPressed variable is handy if you want to quickly check if a random key has been pressed or released as opposed to checking each key's state. For that there is a separate variable called m\_keyStates. The m\_keyStates variable, which is a pointer to an array of Uint8 values, is filled each time a key is pressed and will store a kind of binary snapshot of the keyboard, stating which keys are pressed, and which are not. Each individual element in the array represents a key on the keyboard and can be

queried in the client code as to whether that key is pressed or not. Three button state variables store the pressed state of the three mouse buttons. This is perfect for mouse functions, but for keyboard keys there are too many keys to individually set their state so for that there is a separate keystate array variable.

For each class property variable there is a getter function for when mouse and keyboard states are queried in the client code. There are two IsMouseColliding() functions, both using the current mouse coordinates to create either a AABB box or Sphere around that to check if the mouse collides with the passed bound. This is handy for checking if the mouse cursor is hovering over buttons, spherical things or any other game objects. The IsMouseColliding() could have been passed a generic Bound object, but then dynamic casting is needed to set the dimension each time and that is expensive! Templatizing could also work but the two functions are not exactly identical. Basic function overriding seems to be the best solution here for now. (More TBA) There are two mouse cursor setter functions to set the type and state of the mouse cursor. Using these functions, a specific Windows OS mouse cursor can be created and the cursor may be enabled, disabled, shown or hidden

The Update() function is the core of the Input Manager class. It will process all SDL events that build up on the event queue and will store particular keyboard and mouse property values in the relevant variables. These values can then be used at any time in the client code. This routine will need to be called once per frame in the client code.

**ScreenManager**

This class encapsulates screen management, and is designed to set up all screen and window related matters. It is there to set up all SDL components that will manage the main game window and the canvas it will render to. The SDL renderer is also cleared via this class. The m\_width and m\_height member variables are separate ints because it looks clearer to identify them separately. This class is a Singleton.

The Initialize() routine probably does the most work in the class. It is there to set up the SDL subsystem, and create a game window and a rendering canvas.

The Update() and Draw() routines will clear and render the actual game screen respectively, and the ShutDown() function closes down the SDL subsystems.

**TextureManager**

This class encapsulates a texture manager which will load images from disk and store them as SDL textures in a map, with a string reference to that texture. The images are loaded using SDL's image loading function and generated through SDL's texture creation routine to produce a texture object out of the SDL\_Surface object that was originally loaded. SDL textures make use of the GPU and so are preferred. Currently the images supported are BMP, PNG, JPG, GIF, TGA. <TBA>. To access a particular image inside the map the string reference is needed to sift through the map and find the texture. Textures can also be unloaded from memory individually or in bulk. Furthermore, this class also loads font files and stores them in a similar manner in a separate font map. Fonts and textures are stored together because they are somewhat linked in the original

Handmade engine, and this prevents us from creating a separate Font Manager class. This class is a Singleton.

Two enum types have been created to be used when removing texture images from the map. Either one single specific texture can be removed, or the entire map of textures can be cleared.

To use this class, an object in the client code will need to link to a specific texture or font pointer from within the Texture Manager. The GetFont() and GetTexture() functions are used to request the font and texture pointers respectively and link the data with the external object.

An Output() routine is there for debug purposes only and will print to the console how many font and texture objects are currently stored in the map.

Added a FontSize enum to represent the different sizes to load the fonts in. When loading the fonts, the size can be passed at load time. The MenuKey class can use these sizes to load in the font in the desired resolution into memory. This size enum remains at the lowest level so that any object using fonts does not have to concern itself with sizes, the TM does that.