PARADOX ENGINE

REV: 90414

Table of Contents

Integrated Libraries (x86 and x64)

OpenGL (glew-1.10.0)

- Graphical pipeline for utilization of the gpu, such as rendering objects, shader manipulation
- **Documentation**
  - http://www.opengl.org/sdk/docs/man/
- **Download**
  - http://glew.sourceforge.net/

SDL2-devel-2.0.3

- System pipeline for obtaining window, system, time, video information
- **Documentation**
  - https://wiki.libsdl.org/
- **Download**
  - https://www.libsdl.org/download-2.0.php

SDL2_image-devel-2.0.0

- Image system that allows for loading in various image file formats to a SDL_Surface*
- ICO(Icon)/CUR(Cursor)/BMP, PNM (PPM/PGM/PBM), XPM, LBM(IFF ILBM), PCX, GIF, JPEG, PNG, TGA, TIFF, and XV thumbnail formats
- **Documentation**
  - https://www.libsdl.org/projects/SDL_image/docs/
- **Download**
  - https://www.libsdl.org/projects/SDL_image/

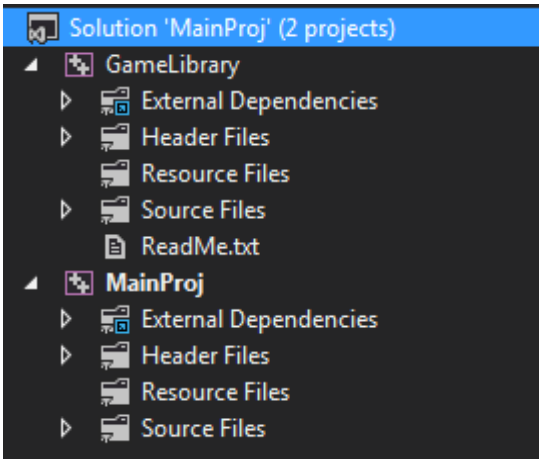
SDL2_ttf-devel-2.0.12

- Font pipeline used for being able to use .ttf fonts, loaded into the system by collecting the SDL_Surface* data generated by the raster
- **Documentation**
  - http://www.libsdl.org/projects/SDL_ttf/docs/
- **Download**
  - https://www.libsdl.org/projects/SDL_ttf/

FMOD (fmodstudioapi-10400) (Not Currently Implemented)

- Sound pipeline for buffering and exerting sound
- **Documentation and Download**
  - http://www.fmod.org/download/fmodstudio/api/Win/fmodstudioapi10400win-installer.exe

Project Information



I separated the projects into two individual pieces. The **GameLibrary** being the .lib / .a portion that encapsulates the Paradox Engine, containing all the pieces that can be utilized in the **MainProj** project, or whatever project you are working in.

The goal I've been trying to do while working on this is to just use the pieces and not have to go back to the engine code to fully integrate it into the particular game zone that is being worked on. There are times when working on something you may encounter a situation where it is necessary to add / modify a particular engine piece. Just try to keep the original behavior the same while also incorporating your particular modification. Such as, recently I needed to also use a GUI Texture2D so I had to modify the Texture2D object to act as such, while still being able to use the default Texture2D object if desired.

The file **main.cpp** within **MainProj** is the aspect which ties the Paradox Engine with the system. This is where you also initialize your particular GameZones for use within the GameWorld. The **InterchangeGameZone(GameWorldState gameWorldState)** function within here is how you tell the system to change to the particular desired GameZone.

Current problems within main:

- main automatically handles SDLK_ESCAPE currently, this is undesirable for custom SDLK_ESCAPE key down behavior
- I'm not convinced with the frame rate method by the frame delay when using SDL_Delay

When I'm working on particular class implementations in their respective .cpp files, I like to put the private functions above the constructor. Under the constructor I put the destructor if there is one, followed by the public functions.

Implemented Classes

GameWorld.h

- The main SDL and OpenGL implementation location and accessors about the particular window that is being used. This also contains information for the various Resources that are being used and the current projectionMatrix that is being used for rendering gpu to screen.
- **main.cpp** is the piece within the MainProj that ties this GameWorld together with the system. See Project Information section for more information.
- **StateValue** are the enums that represent either the previous, current or next state value for manipulating the various GameWorldStates.
- **GameWorldState** by default the Paradox Engine comes with preset GameWorldState values named, Quit, Restart, Logo, Title, Game and Credits. You could define your own unique GameWorldState within **main.cpp** for use for your particular GameZone. GWS_QUIT and GWS_RESTART are the only GameWorldState values that are actively being used within the system's main game loop.
- **Public Variables**
  - **static ResourceManager* resourceManager**
    - Global access to the ResourceManager for loading in particular data, currently only supports SDL_Surface* and ShaderObject*
  - **static Matrix3D* projectionMatrix, invProjectionMatrix**
    - projectionMatrix is the transformation to the desired graphical view, the default right now is Orthographic
- **Public Functions**
  - **GameWorld(const std::string title, int x, int y, int w, int h, Uint32 flags, GameWorldState initState)**
    - **const std::string title**
      - The GameWorld's window title
    - **int x, y, w, h**
      - X position of window, Y position of window, Width of window and Height of window
    - **Uint32 flags**
      - The SDL_WindowFlags to initialize this GameWorld's window with
    - **GameWorldState initState**
      - The initial GameWorldState to start in, the particular GameZone to

GameWorldState is determined in main.cpp, in the InterchangeGameZone(GameWorldState gameWorldState) function.

- **static const int getWindowWidth(); getWindowHeight()**
  - Global access to the window width and height
- **void setGameWorldState(StateValue stateValue, GameWorldState activeState)**
  - Sets the current GameWorldState. StateValue can either be the SV_PREV, SV_CURR, or SV_NEXT to distinguish which state you want to change
- **void setCurrentGameZone(GameZone* gameZone)**
  - Sets the current GameZone to be the desired GameZone that is inputted, see the InterchangeGameZone function within main.cpp
- **SDL_Window* getGameWorldWindow()**
- **SDL_Event* getGameWorldEvent()**
  - Retrieves the current SDL_Event for system events (Keys, Mouse, etc)
- **GameWorldState getGameWorldState(StateValue stateValue)**
- **GameZone* getCurrentGameZone()**
  - Allows access for retrieving information about the current GameZone

---

GameZone.h

- The implementation of a GameZone allows you to create your own particular game level. Within your GameZone various methods are utilized for manipulating and using your Zone.
- **Protected Variables**
  - **std::string zoneName**
  - **float zoneTileWidth, zoneTileHeight, zoneTileRot**
    - The attributes that make up the particular GameZone
  - **GameWorld* gameWorld**
    - Keep track of the GameWorld for access to it's features
  - **Vector3D screenMousePos, worldMousePos**
    - The respective Screen Mouse Position and World Mouse Position
  - **bool leftMouseDown**
    - Is the left mouse button down?
  - **std::list<GameObject*> gameObjectList**
    - Contains a list of all the GameObjects that make up the GameZone
    - Add your GameObject* to this list from within your inherited

GameZone

- **Public Variables**
  - **Matrix3D *viewMatrix, *baseModelWorldMatrix**
    - viewMatrix acts as the camera transformation, baseModelWorldMatrix acts as the transformation representing the grid coordinates of our desired GameZone
- **Virtual Functions**
  - **void LoadZoneData() = 0**
    - The location where you should create all of the initial objects that are desired for the particular GameZone
  - **void InitializeZoneData() = 0**
    - The location where you should initialize your objects and data to their initial points, this is called at the beginning of the loop as well as after a restart occurs
  - **void InputUpdate()**
    - The pre-defined default behavior for this function is determining the screenMousePosition, worldMousePosition and whether the leftMouseDown is toggled. This default behavior can be called / utilized from your own GameZone by doing: GameZone::InputUpdate()
  - **void LogicUpdate(float deltaTime) = 0**
    - The location where the data is updated, make sure to use deltaTime for your various time based calculations and movements
  - **void RenderUpdate() = 0**
    - The location where the rendering of the GameZone takes place
  - **void DeInitializeZoneData() = 0**
    - This is called when the GameWorldState is changing
  - **void UnloadZoneData() = 0**
    - This is called when the GameWorldState is changing to a new GameWorldState
- **Public Functions**
  - **GameZone(GameWorld* gWorld, std::string name, float tileWidth, float tileHeight, float tileDegRot)**
    - **GameWorld* gWorld**
      - The GameWorld* that is being used by the System for the GameZone to be able to access
    - **std::string name**

        - The name of this GameZone
    - **float tileWidth, float tileHeight, float tileDegRot**
        - The tile pixel width, height and rotation
- **T* findGameObject()**
    - Retrieves the first occurrence of a particular class of GameObject within the gameObjectList. Call this by doing: GameObjectClass* yourObject = findGameObject<GameObjectClass>(), GameObjectClass being the GameObject class type you are looking for.
- **std::list<T*> findGameObjects()**
    - Identical to findGameObject<T>(), but retrieves all the GameObjects of the particular class type found.
- **const float getActiveZoneTileWidth(); getActiveZoneTileHeight(); getActiveZoneTileDegRot()**
- **Vector3D ScreenToWorld(const Vector3D& screenPos)**
    - Converts a screen position coordinates to world position coordinates
- **WorldToScreen(const Vector3D& worldPos)**
    - Converts a world position coordinates to screen position coordinates

---

GameObject.h

- Base GameObject class that can be inherited from for creating custom behavior for your desired GameObjects.
- **Protected Variables**
    - **std::string name**
    - **Vector3D position, rotation**
    - **Matrix3D modelMatrix**
        - The matrix that represents the transformation into the coordinate system that represents the GameObject, this is calculated in the LogicUpdate function.
    - **std::list<Component*> objectComponents**
        - A list of all the Component* objects that make up the GameObject, use the attachComponent function to add new Components to this list.
    - **GameZone* gameZone**
        - This provides access to the GameZone this GameObject currently resides in, this can allow you to do things such as access other GameObjects in the GameZone if need be (Whatever the GameZone provides)

- **Virtual Functions**
  - **~GameObject()**
  - **void LogicUpdate(float deltaTime)**
    - The pre-defined behavior of this function is to create the modelMatrix that represents the GameObjects world transformation, followed by updating all the Component* that make up the GameObject.
  - **void Render()**
    - The pre-defined behavior of this function is to call the Render functions of the Component* that make up the GameObject.
- **Public Functions**
  - **GameObject(GameZone* gameZone, std::string name, const Vector3D& position)**
    - **GameZone* gameZone**
      - Allows the GameObject to have access to the GameZone
    - **std::string name**
      - The name of this GameObject
    - **const Vector3D& position**
      - The desired position Coordinates of the GameObject
  - **void attachComponent(Component* newComponent)**
    - Attaches the newComponent to the GameObject's objectComponents list
  - **void setObjectPosition(const Vector3D& nPos); setObjectRotation(const Vector3D& nRot)**
  - **T* getComponent()**
    - Retrieves the first occurrence of a particular class of Component within the objectComponents list. Call this by doing: ComponentClass* yourComponent = getComponent<ComponentClass>(), ComponentClass being the Component class type you are looking for.
  - **std::list<T*> getComponents()**
    - Identical to getComponent<T>(), but retrieves all the Components of the particular class type found.
  - **const std::string& getObjectName()**
  - **const float& getObjectDepth()**
    - Retrieves the Z value that this GameObject resides upon
  - **const Matrix3D& getModelMatrix()**
  - **GameZone& getObjectGameZone()**

- **const Vector3D& getObjectPosition(); getObjectRotation()**

Component.h

- Utilized in conjunction with the GameObject, this allows particular GameObjects to contain Components that contain unique behaviors. Such as, a Texture2D and/or a RigidBody, etc.
- **Protected Variables**
  - **std::string componentName**
  - **Vector3D localPosition; localRotation**
  - **GameObject* parentGameObject**
    - This provides access to the GameObject this Component is attached to.
- **Virtual Functions**
  - **~Component()**
  - **void UpdateComponent(float deltaTime) = 0**
  - **void RenderComponent() = 0**
  - **Component* Clone() = 0**
  - **void attachToGameObject(GameObject* gameObj)**
    - The current use of this function is for attaching custom components to the tiled map object, it is virtual in case a particular component is composed of multiple components and needs to change accordingly.
- **Public Functions**
  - **Component(GameObject* parentObj, std::string name, const Vector3D& position)**
    - **GameObject* parentObj**
      - Allows this Component to access it's associated GameObject*
    - **std::string name**
      - The name of this Component
    - **const Vector3D& position**
      - The localPosition relative to the parentObj
  - **void setComponentPosition(const Vector3D& nPos); setComponentRotation(const Vector3D& nRot)**
  - **const std::string& getComponentName()**
  - **const Vector3D& getLocalPosition(); getLocalRotation()**
    - Retrieves the local position / rotation of the Component
  - **const Vector3D& getWorldPosition(); getWorldRotation()**
    - Retrieves position / rotation relative to the GameObject this

Component is attached to

CustomComponentData

- Used for the integration of custom components into the TiledMapObject
- **Public Variables**
  - **int componentID**
    - The unique custom ID that identifies this particular CustomComponent tile that is used in the TiledMapObject::AttachGridComponentByType function
  - **Component* component**
    - Cloneable Component that will be placed into the location if the particular componentID is encountered when generating the TiledMapObject

---

Matrix3D.h

- A 4x4 matrix implementation utilized as a 3D matrix
- **Public Variables**
  - **RotationAxis**
    - Enum values representing rotation a long the x, y or z axis
  - **float m[4][4]**
- **Public Functions**
  - **void setIdentity()**
    - Sets this matrix to it's identity
  - **void transpose()**
    - Sets this matrix as it's transpose
  - **void concat(const Matrix3D& pMtx)**
    - Concats pMtx onto this matrix
  - **void translate(const Vector3D& translate)**
    - Translates this matrix by the translate Vector3D
  - **void scale(const Vector3D& scale)**
    - Scales this matrix by the scale Vector3D
  - **void rotDeg(float angle, RotationAxis axis); void rotRad(float rad, RotationAxis axis)**
    - Rotates this matrix by angle / rad around the particular RotationAxis, XAXIS, YAXIS or ZAXIS
  - **Vector3D multVec(const Vector3D& pVec)**
    - Multiplies this matrix by pVec return the transformed Vector3D

- **Vector3D getTranslationVector()**
  - Returns what the current translation component this matrix has as a Vector3D
- **void getOpenGL1DFormat(float matrixData[16])**
  - Places this matrix, m[4][4] into matrixData, in column major order (OpenGL's Matrix Format)
- **const Matrix3D& operator*(const Matrix3D& rhs)**
  - Overload multiplier for multiplying left hand Matrix3D with right hand Matrix3D

Vector3D.h

- Implementation of a Vector3D object containing x, y and z values. The z value in the Paradox Engine is currently utilized for just depth layer.
- **Public Variables**
  - **float x, y, z**
- **Public Functions**
  - **void set(const float x, const float y, const float z); void set(const Vector3D& v3ref)**
    - Sets the desired specific Vector3D x, y and z attributes
  - **void negate(); void normalize()**
  - **void scale(const float c)**
  - **void project(const Vector3D& pVec1, const Vector3D& onTo)**
    - Projects pVec1 onto onTo, setting this Vector3D calling the function to the projection
  - **void add(const Vector3D& pVec1, const Vector3D& pVec2); sub(const Vector3D& pVec1, const Vector3D& pVec2)**
    - Adds or Subtracts pVec1 +- pVec2 setting this Vector3D calling the function to the addition or subtraction
  - **void scaleAdd(const Vector3D& pVec1, const Vector3D& pVec2, const float c); scaleSub(const Vector3D& pVec1, const Vector3D& pVec2), const float c)**
    - Adds or Subtracts pVec1 +- pVec2 then scales the resulting value by c, setting this Vector3D calling the function to the scaled addition or subtraction
  - **float length(); float squareLength()**
  - **float distance(const Vector3D& pVec1) const; squareDistance(const**

Vector3D& pVec1) const

- Calculates the distances between this Vector3D calling the function and pVec1

- **float dotProduct(const Vector3D& pVec1) const**
  - Calculates the dot product of this Vector3D calling the function with pVec1
- **const Vector3D& operator+(const Vector3D& rhs) const; operator-(const Vector3D& rhs) const**
  - Addition and Subtraction operators for being able to obtain Vector3D right hand side +- Vector3D left hand side

---

BoxCollider.h

- This box collider utilizes the Separating Axis Theorem for determining if two boxes are interacting with each other. A minimum translation vector can be applied to the colliding object to produce the collision. Publicly Inherits from Component to allow placement onto any GameObject*.
- **Virtual Functions**
  - **~BoxCollider()**
  - **void UpdateComponent(float deltaTime)**
    - Updates and sets the particular Vector3D data that the BoxCollider needs for calculating collision
  - **void RenderComponent()**
    - Currently used for rendering debug information
  - **Component* Clone()**
    - Clones the BoxCollider for use elsewhere if need be (Custom Components)
- **Public Functions**
  - **BoxCollider(GameObject* parentObj, const Vector3D& position, const Vector3D& scale, int colliderColor)**
    - **GameObject* parentObj**
      - This provides access to the GameObject this Component is attached to.
    - **const Vector3D& position**
      - The desired position relative to the parentObj's position.
    - **const Vector3D& scale**
      - The desired scale for this BoxCollider to be.

- **int colliderColor**
  - The desired initial pixel color for the BoxCollider to be when rendering the collision borders.

- **bool isCollidingWithOtherBox(BoxCollider& otherBox)**
  - Determines if this BoxCollider is colliding with the otherBox
- **const Vector3D& getMinTranslation()**
  - Retrieves the minimum translation Vector3D of the collision that took place

---

Debug.h

- Static access class used for accessing and displaying Debug information
- **Public Functions**
  - **static void Log(std::string logText)**
    - Send a message to the debug console
  - **static void DrawLine(const Vector3D& start, const Vector3D& end, const int color)**
    - This function currently doesn't work with the current OpenGL setup
    - This uses the old OpenGL rendering calls for quick debug lines, which requires setup of the old style OpenGL matrices.

FontManager.h

- Used for storing and keeping track of a particular TTF_Font* font, as well as the initialization of the SDL2_ttf pipeline
- **Public Functions**
  - **FontManager(std::string ttfPath, int ptSize)**
    - **std::string ttfPath**
      - The file path directory location of where to load the .ttf font from, .TTF path convention: “data/fonts/fontName.ttf”
    - **int ptSize**
      - The size of the font to load in for usage
  - **TTF_Font* getActiveFont()**
    - Retrieves the TTF_Font* that is associated with the FontManager

Text

- The main component that gives the textual data it's Texture2D component and corresponding data. Publicly inherits from Component allowing placement onto any GameObject.

- **Virtual Functions**
  - **void UpdateComponent(float deltaTime)**
  - **void RenderComponent()**
  - **void Component* Clone()**
- **Public Functions**
  - **Text(GameObject* parentObj, std::string text, const Vector3D& position, bool guiText, TTF_Font* font, int fontColor, int lineLength, ShaderObject* fontShader)**
    - **GameObject* parentObj**
      - The parent GameObject that is associated with this Text Component
    - **std::string text**
      - The textual data that this Text Component will render
    - **const Vector3D& position**
      - If this is a guiText, this is the screen pixel coordinates relative to the GameObject that this Text Component is attached to. Else if this is not a guiText, the world pixel coordinates relative to the GameObject's position.
    - **bool guiText**
      - Is this a GUI Text Component or a Dynamic World Text Component?
    - **TTF_Font* font**
      - The font that this Text Component is made out of. Pass in the font that was allocated in FontManager using: getActiveFont() function.
    - **int fontColor**
      - The 32 bit hex value that represents our color value, ex) 0xFFFFFFFF representing white with full alpha. 0XRRGGBBAA
    - **int lineLength**
      - The line length for how far the textual data should extend before being wrapped
    - **ShaderObject* fontShader**
      - The ShaderObject* that allows the Texture2D portion of the Text to render with it's respective effect. By default use the DefaultTextureShader.

GUITextObject

- A GameObject that contains Text Components, but displays only to the GUI, relative to the viewMatrix. Publicly Inherits from GameObject allowing direct placement and creation of GUI Text.

- **Virtual Functions**
  - **void LogicUpdate(float deltaTime)**
  - **void Render()**
- **Public Functions**
  - **GUITextObject(GameZone* gameZone, std::string text, const Vector3D& position, TTF_Font* font, int fontColor, int lineLength, ShaderObject* fontShader)**
    - **GameZone* gameZone**
      - The GameZone* this GUITextObject belongs to
    - **std::string text**
      - The textual data that makes up this GUITextObject
    - **const Vector3D& position**
      - The screen pixel coordinates for the center point of where the font should be located. (0, 0) Top Left, (ScreenWidth, ScreenHeight) Bottom Right
    - **TTF_Font* font**
      - The font that this Text Component is made out of. Pass in the font that was allocated in FontManager using: getActiveFont() function.
    - **int fontColor**
      - The 32 bit hex value that represents our color value, ex) 0xFFFFFFFF representing white with full alpha. 0XRRGGBBAA
    - **int lineLength**
      - The line length for how far the textual data should extend before being wrapped
    - **ShaderObject* fontShader**
      - The ShaderObject* that allows the Texture2D portion of the Text to render with it's respective effect. By default use the DefaultTextureShader.

ResourceManager.h

- The purpose of the ResourceManager is to store particular loaded data that has possibilities of needing to be loaded again. Currently the only pieces that are saved for future usage are SDL_Surface* and ShaderObject*. These pieces are stored within a std::map where the data access is obtained through the std::string of the filepath location. Currently also, the ResourceManager's data is cleared only when the GameWorld is exiting.

- **Public Functions**
  - **std::map<std::string, SDL_Surface*>* getTexture2DSurfaceResources()**
  - **std::map<std::string, ShaderObject*>* getShaderObjectResources()**
  - **static SDL_Surface* LoadSDLSurface(std::string filePath)**
    - Global access for loading in a particular SDL_Surface* for both retrieval and storage of the particular valid SDL_Surface*.
    - The SDL_Surface* is identified within it's std::map through the filePath
  - **static ShaderObject* LoadShaderObject(std::string name, std::string v_filePath, std::string f_filePath)**
    - Global access for loading in a particular ShaderObject* for both retrieval and storage of the particular valid ShaderObject*
    - The ShaderObject* is identified within it's std::map through the name

---

RigidBody.h

- Publicly inherits from Component, such that this can be attached to a particular GameObject to give it physical kinematic and forced based behaviors. The RigidBody also manipulates the GameObject that it is attached to through the various provided functions.
- **Virtual Functions**
  - **void UpdateComponent(float deltaTime)**
  - **void RenderComponent()**
  - **Component* Clone()**
- **Public Functions**
  - **RigidBody(GameObject* parentObj, const Vector3D& vel, const Vector3D& accel, float mass, float angVel, float angAccel, float inert)**
    - **GameObject* parentObj**
      - The parent GameObject that is associated with this RigidBody Component
    - **const Vector3D& vel**
      - The initial velocity for this RigidBody to start with
    - **const Vector3D& accel**
      - The initial acceleration for this RigidBody to start with
    - **float mass**
      - The mass of this RigidBody
    - **float angVel**
      - The angular velocity this RigidBody starts with

- **float angAccel**
  - The angular acceleration this RigidBody starts with
- **float inert**
  - The inertia of this RigidBody

- **void ApplyVelocity(const Vector3D& vel)**
  - Applies the desired velocity to the RigidBody
- **void ApplyForce(const Vector3D& force)**
  - Applies the desired force to the RigidBody
- **void ApplyTorque(float torque)**
  - Applies the desired torque to the RigidBody
  - Positive torque rotates to the right
  - Negative torque rotates to the left
- **float getSqrVelocity()**
  - Retrieves the current square velocity of the RigidBody

ShaderObject.h

- **Public Functions**
  - **ShaderObject(std::string shaderName, std::string vertexShaderPath, std::string fragmentShaderPath)**
    - **std::string shaderName**
      - The name of the ShaderObject, make sure to use the same ShaderObject name globally. This allows the ResourceManager to load the correct ShaderObject if it has already been loaded before.
        - There is also a note in ResourceManager.h about trying to solve this problem of having to keep track of the same name globally
    - **std::string vertexShaderPath**
      - The file path of where the vertex shader file is located, shader path convention: “shaders/v_shaderName.vert”
    - **std::string fragmentShaderPath**
      - The file path of where the fragment shader file is located, shader path convention: “shaders/f_shaderName.frag”
  - **GLuint getShaderProgram()**
    - Retrieves the OpenGL shader program that represents this ShaderObject* for usage.
  - **GLuint getCompiledStatus()**
    - Returns either GL_FALSE or GL_TRUE if the ShaderObject compiled

successfully.

SoundObject.h (Not Currently Implemented)

- This will be the piece which will allow a GameObject to emit a particular sound upon whichever situation's taking place.

Texture2D.h

- This is the main Component that allows the rendering of Textures within the Paradox Engine. The Texture2D publicly inherits from the Component where it can behave as either a GUI Texture2D or a Dynamic World Texture2D. The Texture2D Component also can be loaded in as a Sprite Sheet, consisting of multiple frames.
  - The GUI Texture2D stays relative to the screen space. When the Texture2D is in this mode it's texture uv wrap mode is set to GL_CLAMP_TO_EDGE to eliminate a pixel border glitch.
  - The Dynamic World Texture2D stays relative to the world space. When the Texture2D is in this mode it's texture uv wrap mode is set to GL_REPEAT.
- **Virtual Functions**
  - **~Texture2D()**
  - **void UpdateComponent(float deltaTime)**
  - **void RenderComponent()**
  - **Component* Clone()**
- **Public Functions**
  - **Texture2D(GameObject* parentObj, std::string filePath, const Vector3D& position, bool guiTexture, int tint, ShaderObject* shader, float xPixelScale, float yPixelScale)**
    - **GameObject* parentObj**
      - The GameObject that is associated with this Texture2D.
    - **std::string filePath**
      - The file path location for loading this particular Texture2D, the Texture2D path convention is: “gfx/imageName.png”.
      - I have only testing this with .png files, but with modification it can load the other formats support by SDL_image.
    - **const Vector3D& position**
      - If this is a guiTexture, the screen pixel coordinates relative to the GameObject. Else the world pixel coordinates relative to the

GameObject.

- **bool guiTexture**
  - Is this Texture2D a GUI Texture2D or a Dynamic World Texture2D.
- **int tint**
  - The 32 bit hex value representing the color tint for this Texture2D to contain. Ex) 0xFFFFFFFF in the form: 0xRRGGBBAA
- **ShaderObject* shader**
  - The ShaderObject* that allows the Texture2D portion to render with it's respective effect. By default use the DefaultTextureShader.
- **float xPixelScale**
  - The desired x pixel width for this object to scale and render as.
- **float yPixelScale**
  - The desired y pixel height for this Texture2D to scale and render as.

- **Texture2D(GameObject* parentObj, std::string filePath, const Vector3D& position, bool guiTexture, float xFrameSize, float yFrameSize, int xFrame, int yFrame, int tint, ShaderObject* shader, float xPixelScale, float yPixelScale)**
  - **GameObject* parentObj**
    - The GameObject that is associated with this Texture2D.
  - **std::string filePath**
    - The file path location for loading this particular Texture2D, the Texture2D path convention is: “gfx/imageName.png”.
    - I have only testing this with .png files, but with modification it can load the other formats support by SDL_image.
  - **const Vector3D& position**
    - If this is a guiTexture, the screen pixel coordinates relative to the GameObject. Else the world pixel coordinates relative to the GameObject.
  - **bool guiTexture**
    - Is this Texture2D a GUI Texture2D or a Dynamic World Texture2D.
  - **float xFrameSize**
    - The x pixel width that makes up an individual frame.
  - **float yFrameSize**
    - The y pixel height that makes up an individual frame.
  - **int xFrame**
    - The initial x frame that this Texture2D starts as

- **int yFrame**
  - The initial y frame that this Texture2D starts as
- **float xPixelScale**
  - The desired x pixel width for this object to scale and render as.
- **float yPixelScale**
  - The desired y pixel height for this Texture2D to scale and render as.

- **Texture2D(GameObject* parentObj, SDL_Surface* sdlSurface, const Vector3D& position, bool guiTexture, int tint, ShaderObject* shader, float xPixelScale, float yPixelScale)**
  - This does the same thing as Constructor #1, but loads directly in from an SDL_Surface*. This currently does not load to and from the ResourceManager's SDL_Surface* std::map.
    - The reason this doesn't load to and from the ResourceManager is because the SDL_Surface* in this case is usually temporary and dynamic, such as the Text Component usage.
- **Texture2D(GameObject* parentObj, std::string filePath, const Vector3D& position, bool guiTexture, float xFrameSize, float yFrameSize, int xFrame, int yFrame, int tint, ShaderObject* shader, float xPixelScale, float yPixelScale)**
  - This does the same thing as Constructor #2. See Constructor #3.
- **void setShaderObject(ShaderObject* desiredShader)**
  - Sets this Texture2D to use a particular ShaderObject, load from ResourceManager
- **void setTextureFrame(int xFrame, int yFrame)**
  - Sets this Texture2D x and y frame coordinates
- **int getCurrentFrameX()**
  - Gets the current x frame coordinate
- **int getCurrentFrameY()**
  - Gets the current y frame coordinate
- **int getXTotalFrames()**
  - Gets the total amount of x frames that make up this Texture2D
- **int getYTotalFrames()**
  - Gets the total amount of y frames that make up this Texture2D
- **int getImageWidth()**
- **int getImageHeight()**
- **int getColorTint()**

- Gets the 32 bit hexadecimal color representation. Ex) 0xFFFFFFFF in the form 0xRRGGBBAA. Use the Utilities static function, Utilities::CalculateRGBAColors to get the RGBA color values.

- **bool isGuiTexture()**
  - Is this a GUI Texture2D?
- **ShaderObject& getActiveShader()**

TiledMapObject.h

- The Paradox Engine TiledMapObject is inherited by the GameObject class, allowing for usage and placement into a particular GameZone. Using the Tiled Map Editor program (tiled.exe), this GameObject allows TiledMapObject GridObject Components to generate in by reading in the exported Tiled Map Editor .txt file. The TiledMapObject file path convention is: "data/maps/exportedMapName.txt".
  - Currently the TiledMapObject expects the "gfx" folder when searching for it's particular Tile Sheets, to be two directories back. See the private member function: TiledMapObject::ReadinTileData.
- **Virtual Functions**
  - **~TiledMapObject()**
  - **void LogicUpdate(float deltaTime)**
  - **void Render()**
- **Public Functions**
  - **TiledMapObject(GameZone* gameZone, std::string filePath, const Vector3D& position, const TileType** tileTypes, int customTileCount, std::list<CustomComponentData>& customMapComponents, float xMapPixelWidth, float yMapPixelHeight)**
    - **GameZone* gameZone**
      - Allows the TiledMapObject to have access to the GameZone it is in
    - **std::string filePath**
      - The file path location for loading in the particular exported Tiled Map Editor .txt file. The TiledMapObject file path convention is: "data/maps/exportedMapName.txt".
        - Currently the TiledMapObject expects the "gfx" folder when searching for it's particular Tile Sheets to be two directories back. See the private member function: TiledMapObject::ReadinTileData.

- **const Vector3D& position**
  - The desired world coordinates for where to center this TiledMapObject
- **const TileType** tileTypes**
  - This is utilized as a TileType* to an array of TileType*. The particular array of TileTypes that this points to represents the TileType IDs that identify the particular tile gfx type. Premade TileTypes that are included with the ParadoxEngine are: TEXTURE_TILE and COLLISION_TILE
- **int customTileCount**
  - How many customMapComponent objects are being added to the TiledMapObject system. If you set this to zero, the std::list customMapComponents value can be ignored
    - Currently since this is a reference it can't be ignored, but it should be allowed to be ignored. There may even be a better way to do this customTileCount check.
- **std::list <CustomComponentData>& customMapComponents**
  - The std::list<CustomComponentData>& that contains the customMapComponents that will be Cloned and then inserted into the TiledMapObject Component std::list when the particular custom ID is encountered when generating the TiledMapObject.
- **float xMapPixelWidth**
  - The x pixel width of each TiledMapObject GridComponent that will be used as the xPixelScale for each TiledMapObject Texture2D
- **float yMapPixelHeight**
  - The y pixel height of each TiledMapObject GridComponent that will be used as the yPixelScale for each TiledMapObject Texture2D

- **std::vector<GridObject**>* getTiledGridObjects()**
  - The GridObject** that these point to are utilized as two dimensional arrays of size TiledMapObject::mapHeight by TiledMapObject::mapWidth. If no GridObject has been created within the desired index values, NULL is returned.
- **int getMapWidth()**
  - Retrieves the map width that this TiledMapObject represents.
    - Ex) 20 x 20 = 400 GridObjects
- **int getMapHeight()**

- Retrieves the map height that this TiledMapObject represents
  - Ex) 20 x 20 = 400 GridObjects
- **float getMapTileWidth()**
  - Retrieves the x pixel width that each TiledMapObject GridComponent represents
- **float getMapTileHeight()**
  - Retrieves the y pixel height that each TiledMapObject GridComponent represents

GridObject

- This is the class object that encapsulates the data that makes up the TiledMapObject
- **Public Variables**
  - **float xCoord, yCoord, zCoord**
  - **GridObject* adjGridObjects**
    - Currently not being utilized nor initialized, but will be
  - **Component* gridComponent**

TileType

- This is the enum of data that represents the pre-defined TileTypes that the Paradox Engine currently has implementations for.
- It currently is possible to create your own Component objects with a unique custom ID that isn't one of these pre-defined TileType values. See the constructor of TiledMapObject for more information

---

Utilities.h

- Static access class used for accessing and using particular Utilities that help with particular purposes, ideally the functions here can be used for multiple purposes in more than one object.
- **Public Functions**
  - **static void CalculateRGBAColors(float colorData[4], int color)**
    - Calculates the RGBA float values of the 32 bit hex color value, ex) 0xFFFFFFFF in the form 0xRRGGBBAA. The individual R, G, B, A values are placed into the colorData[4] in the range [0, 1].
  - **static std::string ParseAndCollectFile(std::string filePath)**
    - Parses a particular file at filePath and returns the entire file as an std::string
  - **static std::string ParseFromCollectTo(std::string text, const char start =**

'\0', int startDelay = 1, const char end = '\0')

- Parses a particular std::string returning the parsed text starting from the start character and ending and the end character.
- **std::string text**
  - The std::string to parse
- **const char start = '\0'**
  - By default the start character '\0' starts the parse at the first character encountered, but a particular character can be defined for when to start the parse
- **int startDelay = 1**
  - Additionally you can set a startDelay for how many times the start character needs to be encountered before beginning the parse
- **const char end = '\0'**
  - The end character can be used for when to stop the particular parse, by default this will end when the parse has reached the end of the std::string

---

VertexArrayObject.h

- The Paradox Engine VertexArrayObject is an encapsulation of the OpenGL Vertex Array Object rendering system. Where this keeps track of the particular Vertex Buffer Data that this VertexArrayObject is composed of. The Vertex Buffer Data is data that represents information such as: The mesh coordinate information, texture coordinate information, color coordinate information, normal coordinate information and I'm sure there are other usages even beyond this
- **Public Functions**
  - **CreateVertexBuffers(int bufferCount, GLfloat** buffers, int* bufferSizes, GLenum usage)**
    - This function creates the OpenGL Vertex Buffer Objects for usage within the Paradox Engine.
    - **int bufferCount**
      - How many Vertex Buffer Objects will be loaded for usage in the Vertex Array Object
    - **GLfloat** buffers**
      - This is utilized as a GLfloat* to an array of GLfloat* containing the actual Vertex Buffer Object data

- **int* bufferSizes**
  - This is utilized as an array of ints of size bufferCount containing the byte size of each buffer within buffers
- **GLenum usage**
  - This value has to be either GL_STATIC_DRAW or GL_DYNAMIC_DRAW

- **void InitializeVertexArray(GLint* varPieceCount)**
  - This function initializes the OpenGL Vertex Array Object with the particular Vertex Buffer Objects for it to maintain and keep track of.
  - **GLint* varPieceCount**
    - Array of GLints that contain how many values make up each variable piece, the size of this is the same size of how many Vertex Buffer Objects are being used with this Vertex Array Object
- **GLuint getVertexArrayObjectID()**
  - This function retrieves the OpenGL Vertex Array Object ID that represents this Paradox Engine's encapsulated Vertex Array Object
- **void ChangeVertexBuffer(int index, GLfloat* newBuffer, int bufferSize, GLenum usage)**
  - This function changes a particular Vertex Buffer Object's data at the index location of the order in which they were generated into the Vertex Array Object
  - **int index**
    - The index location for which Vertex Buffer Object data to change, it is up to the user to know which index they want to access. Ex) mesh coordinate information
  - **GLfloat* newBuffer**
    - The new data to be placed into the desired Vertex Buffer Object
  - **int bufferSize**
    - The byte size of newBuffer
  - **GLenum usage**
    - This value has to be either GL_STATIC_DRAW or GL_DYNAMIC_DRAW
- **int getBufferDataSize(int index)**
  - This retrieves the particular Vertex Buffer Object byte size at the particular index. It is up to the user to know which index they want to access.