TECHNICAL OVERVIEW

Major Systems

Graphics

Physics

Audio

GameState Manager

GameObject Manager

JSON Serialization

Behaviors

Messaging

Major Technologies

OpenGL 4.5

FMOD Low-Level API

Components

Transform

RigidBody

Colliders

Sprite:

- Sprite
- UISprite
- CircleSprite
- UICricleSprite

Background

Camera

ForceEmitter

ParticleEmitter

Behaviors

CODING METHODS

File Locations:

```
/
      data/
            glsl/
            img/
                  bg/
                  spr/
            json/
            physics/
            sounds/
                  effects/
                  music/
      src/
            Audio/
            Behaviors/
            Engine/
            GameObject/
            GameStateManager/
            GameStates/
            Graphics/
            Input/
            Messaging/
            Physics/
            Trace/
            Transform/
            Util/
```

Source Control:

GIT: `https://git.digipen.edu/projects/datalust`

Style Guide:

All identifiers should be snake case. Ex:

```
int counter_value;
float my_float;
GameState game_state;
```

All Namespaces and User Defined Types should be camel case with a leading capital. Ex:

```
namespace MyNamespace {
  class MyClass {
    public:
      void myMethod();
    private:
      int my_data_;
  };

  struct MyStruct {

  };
```

```
    enum class MyEnum {
      FirstValue,
      SecondValue
    };
  }
```

All Functions should be camel case with a leading lowercase. Ex:

```
  void myFunction() {

  }

  class MyClass {
    public:
      void myFirstMethod();
      void mySecondMethod();
    private:
  };
```

All private class members should have a trailing underscore. Ex:

```
  class MyClass {
    private:
      int internal_counter_;
      float value_;
      void myPrivateMethod_();
  }
```

All modules should be enclosed in a Namespace.

All files should have Doxygen comments.

Block comments should be used used for File, Namespace, Function and User Defined Type headers. Block comments should use the double-star format, should have the comment delimeters on their own lines, and the body of the comment should be indented one level from the delimeters. Ex:

```
    /**
      \breif This is a function.
    */
    void MyFunction ();
```

Namespace, Function and User Defined Type headers should only be present in header files.

Line comments should be used for documenting member variables. Line comments should the triple-slash format, and should be placed on the same line as the member variable they are documenting. Ex:

```
    /**
```

```
      \brief My Enum.
    */
    enum class MyEnum {
      FirstValue,  ///< The first value.
      SecondValue, ///< The second value.
      ThirdValue   ///< The third value.
    };
```

File headers must include, at minimum, the filename, the author's name, a brief and long description, and the DigiPen copyright information. Ex:

```
    /**
      \file   GraphicsTypes.hpp
      \author Samuel Cook

      \brief Types used by the Graphics Engine.

      A collection of enums and POD-structs that are used in the Graphics engine's
interface and by
      the private implementation.

      Copyright 2017 DigiPen Institute of Technology.
    */
```

Comments within functions describing implementation details should NOT be Doxygen comments.

DEBUGGING TOOLS

Output-only console

Debug draw for physics colliders

Trace logging to file

GRAPHICS OVERVIEW

- Sprites are being rendered by instancing a 1x1 quad centered at the origin.
- Parallax is applied to backgrounds in shaders.
- OpenGL 4.5:
  - Using GLSL vertex and fragment shaders.
- Texture Loading:
  - Images are being opened with SFML and then being manually loaded onto the graphics card.
- Graphics interfaces with the rest of the game through components.

Graphics Internals

- Window (Window.cpp/.hpp)
  - wrapper around SFML window. Needs refactoring.
- Renderer (Renderer.cpp/.hpp)
  - Performs OpenGL Rendering. Needs refactoring.
- Graphic (GraphicBase.cpp/.hpp)
  - Common base class for all visible graphic elements. Implements common properties.
- Mesh (Mesh.cpp/.hpp)
  - Wrapper around the unit quad mesh. Each instance has it’s own uv coordinates. Needs refactoring.
- Texture (Texture.cpp/.hpp)
  - wrapper around OpenGL textures. Needs refactoring.

GameObject Components

Sprite (Sprite.cpp/.hpp)

Positioned using the GameObject’s Transform component.

Can be optionally textured and animated.

Index

which subset of the texture to map onto the sprite. Indexes start at zero in the top-left of the sprite, and increase left-to-right, top-to-bottom. e.g.

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

CircleSprite (Sprite.cpp/.hpp)

Same as Sprite, but the image is mapped to an ellipse instead of a rectangle.

UISprite (Sprite.cpp/.hpp)

Same as a sprite, but the Transform component is interpreted to be in camera space instead of world space

UICircleSprite (Sprite.cpp/.hpp)

Same as UISprite, but the image is mapped to an ellipse instead of a rectangle.

BackgroundImage (Background.cpp/.hpp)

Renders an Image as a screen-sized quad with a tilling, parallax background.

ParallaxFactor

How much parallaxing is applied to the background; how much the background moves when the camera scrolls.
$p = 0$ – background does not move, appears fixed in the viewport
$0 < p < 1$ – background moves less than sprites
$p = 1$ – background appears to move with sprites (default)
$p > 1$ – background moves more than sprites

Camera (Camera.cpp/.hpp)

This component allows the GameObject to be used as a viewport, according to it’s Transform component.

Graphic Properties (common to Sprite, CircleSprite, UISprite, UICircleSprite, and BackgroundImage)

Depth

depth-order used for depth sorting. Uses left-handed z-axis ordering, i.e. objects with lager depth values appear behind objects with lower depth values.

Alpha
: Opacity of the graphic. 1 is fully opaque and 0 is fully transparent.

BlendColor
: Color used for optional blending. Uses normalized float RGB.

BlendAlpha
: Amount to blend the BlendColor with the Texture. 1 is full BlendColor, 0 is full Texture.

BlendMode
: Mode used to blend the BlendColor with the texture. The following blend modes are supported and function the same as in Photoshop:
    - None
    - Mix
    - Multiply
    - Screen
    - Overlay

PHYSICS OVERVIEW

Main:
- Physics.cpp/hpp

Physics Namespace Global Variables:
- PhysicsConstants.hpp

Physics System Components:
- BoundArea.cpp/hpp
- CollisionDetect.cpp/hpp
- ContactResolver.cpp/hpp
- ForceGenerator.cpp/hpp
- ForceRegistry.cpp/hpp
- Particle.cpp/hpp

GameObject Components:
- Colliders.cpp/hpp
- ForceEmitter.cpp/hpp
- ParticleEmitter.cpp/hpp
- RigidBody.cpp/hpp

Main:

- Contains the main loop of the physics system.

Physics Namespace Global Variables

- Contains
  - Physics Constants
  - List of Force Types
  - List of Bound Types

Physics System Components:

BoundArea

- Contains boundary boxes.
- Aggregate of Colliders

CollisionDetect

- Contains functionality for detecting collisions.
- Generates contact lists.

ContactResolver

- Contains functionality for resolving contacts.
- Takes generated contact lists and resolves.
  - Resolution sends Messages to the Messaging system.

ForceGenerator

- Contains Forces that can be applied to RigidBodies
  - IE: Gravity/Drag
- Aggregate of ForceRegistry
- Interface functions
  - ForceGenerator* makeForceGenerator(ForceType type, glm::vec2 vec1, glm::vec2 vec2, float val1, float val2);
    - Creates a force generator.

ForceRegistry

- Interface for storing existing pairs of Forces and RigidBodies.

- Interface functions
    - void add(RigidBody* body, ForceGenerator* forceGen);
        - ☐ Adds a Force to an existing RigidBody.
        - ☐ Force must be created by makeForceGenerator function before being used.
    - void remove(RigidBody* body, ForceGenerator* forceGen);
        - ☐ Removes a specified Force from a RigidBody.
    - void removeByType(RigidBody* body, ForceType type, bool emitter);
        - ☐ Removes all of a Force Type from a RigidBody.
    - void clear()
        - ☐ Removes all existing forces

<u>Particle</u>

- Contains particles
- Aggregate of ParticleEmitter

<u>**GameObject Components:**</u>

<u>Colliders</u>

- Contains a list of Bounds attached to a single RigidBody
- Interface functions
    - void add(BoundType type, Transform* body, float xHalfsize, float yHalfsize, float xOffset = 0, float yOffset = 0);
        - ☐ Adds a Bound to the Collider.
    - void removeType(BoundType type);
        - ☐ Removes all Bounds of given type from Collider
    - bool raycastFromTrans(glm::vec2 direction, float length);
        - ☐ Raycasts from the transform of the object a given distance.
    - float getTop(void) const;
        - ☐ Returns the top limit of all Bounds
    - float getBottom(void) const;
        - ☐ Returns the bottom limit of all Bounds
    - float getRight(void) const;
        - ☐ Returns the right limit of all Bounds.
    - float getLeft(void) const;
        - ☐ Returns the left limit of all Bounds

<u>ForceEmitter</u>

- Contains a single Force, list of RigidBodies affected by the Force, and Range.
    - Range differs based on emitter shape
        - ☐ Box
        - ☐ Circle
    - ForceType and values can be set
- Interface functions
    - void setForce(ForceType force, ForceGenerator* forceGen);
        - ☐ Sets the Emitter to emit given Force
    - void setRadius(float radius);
        - ☐ Set EmitterCircle radius
    - float getRadius(void) const;

        - Get EmitterCircle radius
    - void setHalfsize(glm::vec2& halfsize);
        - Set EmitterBox halfsize
    - glm::vec2 getHalfsize(void) const;
        - Get EmitterBox halfsize

ParticleEmitter

- Incomplete, please ignore

RigidBody

- Main physics components.
    - No functionality of the Physics System will work on a object without a RigidBody.
- Contains
    - Mass
    - Velocity
    - Damping
    - Acceleration
    - Restitution (Bounciness)
- Interface functions
    - All variables listed above have getters and setters
    - bool forcesAreActive(void);
        - Check if Forces are active on this RigidBody
    - void forcesActivate(void);
        - Activates Forces on RigidBody
    - void forcesDeactivate(void);
        - Deactivates Forces on RigidBody
        - A deactivated RigidBody will not be affected by any force

BEHAVIORS OVERVIEW

Behaviors implemented as components

- BoundryBehavior
  - implements logic for the left and right barriers that keep players in until they reach a certain velocity
- CameraBehavior
  - implements the camera scrolling logic, as well as the death logic
- HUDBehavior
  - implements the logic of the HUD indicators showing player's current level of damage
- MenuBehavior
  - implements the logic for the main menu
- PlatformBehavior
  - implements the logic for semi-solid platforms
- Player1Behavior
  - implements the logic for Player 1, should be combined with Player2Behavior and refactored into PlayerBehavior next semester.
- Player2Behavior
  - implements the logic for Player 2, should be combined with Player1Behavior and refactored into PlayerBehavior next semester.
- PunchBehavior
  - implements the logic for Punches, which are implemented as separate objects with their own logic that are spawned by players when they attack

AUDIO OVERVIEW

Audio:

We are using the low level API for the audio manager. The audio manager creates sound objects given parameters read in from a file and stores them in an unordered map along with their name to search by.

Functions include:

Audio Manager:

- **Init()**
- **Update(float dt)**
- **Shutdown()**
- **Load()**
  - Checks manifest file and loads sounds in
- **Unload()**
  - Releases all FMOD sounds and destroys all sound objects and the map
- **PlaySound(const char* name, unsigned int delay = 0U)**
  - Looks up and plays a sound by name with a given delay (defaulted to 0), if the sound is not found, play the default sound
- **PauseSound(const char* name)**
  - Pauses a specific sound
- **UnpauseSound(const char* name)**
  - Unpauses a specific sound
- **StopAllSound()**
  - Stops all currently playing sounds
- **StopEffects()**
  - Stops all currently playing effect sounds
- **StopMusic()**
  - Stops all currently playing music sounds
- **Getters & Setters for:**
  - **Effects Volume, Pitch, Frequency**
  - **Music Volume, Pitch, Frequency**
  - **Master Volume, Pitch, Frequency**
    - Gets, sets, and updates values of all or specific sounds
- **GetOutputRate()**
  - Gets the output rate of the system in samples
- **PlayRandomSound(std::vector<std::tuple<std::string, unsigned int, float>> sounds)**
  - Given several sounds, their delay, and their probability, play a random one and return which one was played
- **PlaySequence(std::vector<std::pair<std::string, unsigned int>> sequence, bool relative)**

  - Given several sounds and their delay (from previous sound's start if relative is true, or from the start of the sequence if relative is false), play them in sequence
- **PlayRandomSequence(std::list<std::vector<std::tuple<
  std::string, unsigned int, float>>> sequence, bool relative)**
  - Given several groupings of several sounds and their delay (same relative rules) and their probability, chooses 1 sound from each group and play them in sequence.

Sound:

- **Sound(const char* name, int type, bool looping)**
  - Create a new sound object
- **~Sound()**
  - Destroy a sound object
- **Play(unsigned int delay)**
  - Play a sound with a given delay (defaulted 0)
- **Pause()**
  - Pause the sound
- **Unpause()**
  - Unpause the sound
- **Setters for:**
  - **Pitch, volume, frequency, type, and name**
    - Sets and updates values. No getters because the user should not access these

GAME STATES OVERVIEW

Main:

GameStateManager.cpp

GameStates:

GameState.cpp/h

Main:

- The GameStateManager contains a std::stack of the Gamestates, as well as the ability to Change, pop, push, quit, and restart states
- A level serializer used to read in all of the Levels and stores them in the
- ☐ array, this is used to easily index into levels and create new ones when needed

functions:

- ☐ `GameStateManager();`
  - constructor for the GameStateManager, puts a nullptr as the first member of the stack for safety/sanity checking. It also sets the isUpdating flag to be false
- ☐ `~GameStateManager();`
  - destructor for the GameStateManager, calls the Quit function, and then de-alocates the data used for the stack
- ☐ `bool Init();`
  - Init function pushes the initial GameState onto the stack and calls the states Init function, this is separate from the constructor so that the manager can be created without data
- ☐ `void Update(float dt);`
  - Update function for the GameStateManager class updates the GameState on the top of the stack (Stack manipulation is guaranteed not to occur inside the update function)
- ☐ `void Quit();`
  - Pop’s GameStates off of the state until it reaches the nullptr at the bottom
- ☐ `void ResetState();`
  - Restarts the State on the top of the stack by Calling the states Shutdown and Init functions
- ☐ `void ChangeState(const int level);`
  - changes the state by poping the current state and pushing a new state by indexing into the vector of levels using the int passed in
  -
- ☐ `void ChangeState`
  - Overloaded change state function that changes to the next level in the vector
- ☐ `void PushState(const int new_state);`

· pushes a new game state onto the top of the stack and then calls th the state Init function

☐ `void PopState();`
  · Calls shutdown on the state that is on the top of the stack, and then removes it from stack

☐ `GameState* CurrentState();`
  · returns the GameState at the top of the stack

☐ `bool IsRunning();`
  · determines whether the GameStateManager is running, if it is it returns true, else it returns false

GameState Components:

GameObjectManager

- Slot map that contains and updates all of the GameObjects that belong to the state has
- A std::list of ObjectId's (Indexes into the SlotMap)

filename

- A filename which represents the json file the state loads objects from

functions:

☐ `void Init();`
  ◦ Loads a GameState from a json file and creates a GameObjectManager, then registers any messages needed

☐ `void Update(float dt);`
  ◦ Calls the GameObjectManager's Update function, determines whether the game should be changed or reset

☐ `void Shutdown();`
  ◦ Shuts down the GameObjectManager, Unregisters all of the messages registered in the Init Function and stops all music

GAME OBJECTS OVERVIEW

Main:

GameObject.cpp/h

Main:

- Contains an std::unordered_map of components (transform, sprite, etc…)
- Contains Objects Name and type
- **functions:**
  - `GameObject(std::string type, std::string name = "");`
    - constructor for the GameObject, Creates a GameObject with the Type and Name passed in (The name defaults to an empty string)
  - `GameObject(const GameObject& other);`
    - Copy constructor for the GameObject class copies all of the components and private data to a new object
  - `GameObject(GameObject&& o) noexcept;`
    - Move constructor for the GameObject moves an object and its components
  - `template<typename T>GameObject& Add(T* component);`
    - Add a component to the GameObject
  - `template<typename T>const T* Find() const & non const;`
    - Find a component attached to the GameObject
  - `~GameObject() noexcept;`
    - Destructor for the GameObject class, deletes all of the components
  - `void Destroy();`
    - Flag a game object for destruction, This is to avoid game objects being destroyed while they are being processed.
  - `void ChangeState(const int level);`
    - changes the state by poping the current state and pushing a new state by indexing into the vector of levels using the int passed in
  - `bool IsDestroyed();`
    - Check whether a game object has been flagged for destruction.
  - `void Update(float dt);`
    - Updates any components attached to the GameObject

MULTIPLAYER OVERVIEW

Non-networked 2-player versus.

2 players, 1 keyboard

2 players, 2 game pads