MULTIMEDIA DESIGN

Team Multimedia

CONTENTS

OVERVIEW

This document identifies and explains all tasks that the Multimedia team will be expected to complete for the project. These tasks are broken into four categories; design, sound, programming, and art. The bulk of the work will come from asset creation and integrating the assets within the game. In additional to work breakdown, this document also includes the design work completed by the team.

TEAM

One team will be working on multimedia throughout the project in order to have consistency and harmony between components. This team will be broken down roughly follows:

Melvin Loho && **Jonathan Chu** – Major Programming

Rhea Lauzon (Team Leader) – Art & Audio/Visual direction

Lewis Scott – Sound & Minor Programming

WORK BREAKDOWN

MELVIN LOHO, LEWIS SCOTT, & RHEA LAUZON

WORK BREAKDOWN

The list of tasks to be performed by the multimedia have been broken down into sub-sections to identify the area of development they belong to. Any tasks that are unclear or vaguer have been given a short description for clarification.

PROGRAMMING

By Melvin Loho

This section will list and explain all of the ideas / concepts / classes / engines required to store, manage, represent, render and output the assets we produce.

GRAPHICS

SCENE GRAPH / GRAPHICAL HIERARCHY USING GAME OBJECTS [GO]

Every game entity, whether it is seen as a sprite, a text object, a combination of both or by something else, needs to be encapsulated in a Game Object. GOs will serve the purpose of helping the things that they hold to exist in the environment correctly (visibility/transformations) and hierarchically. A GO can be added into another GO, thus forming a parent-child bond.

9-PATCH IMAGE SUPPORT

The backgrounds of GUI elements can either be made of plain colors or a texture. Using a texture that can be scaled infinitely will be very beneficial as it will introduce a lot of flexibility and ease. One way to do this is to use the 9-patch image concept, where an image will be split into 9 pieces in order to stretch the areas that can be stretched without making any undesirable stretchings or pixelations.

PARALLAX SCROLLING

A pseudo-3D effect where background elements scroll slower than foreground elements. We can manipulate the scene graph’s transformations hierarchy logic in order to give us this effect.

GRAPHICAL ENGINES

APPWINDOW - SCENES RELATIONSHIP

An instance of a game will have an instance of an AppWindow. It is an extension to SFML's sf::RenderWindow that will provide easy access to manipulate settings of the window. It holds a vector of Scenes that can be added and removed at any time. The main game loop will reside in the AppWindow and there are 5 main parts that a Scene will be able to override and customize:

+ loading
+ unloading
+ event handling
+ updating
+ rendering

A Scene will act as a part of the game that you can see and interact with (equivalent to Android's Activities).

PARTICLE SYSTEM

Particle systems are containers in which particles are made, manipulated, built, and rendered. They should be extended for uses like making fireballs or smoke emitters.

A particle system holds many variables to control its particles with:

+ lifetime
+ colour over life
+ size over life
+ emitter position
+ gravity
+ texturing
+ particle rotation
+ particle velocity
+ collisions

TILEMAP ENGINE

This will be able to read/parse in data in order to build a map that uses a single texture (tileset) for creating the environment with.

RENDERING

SPRITE BATCHING

A rendering technique that increases rendering performance by batching similar sprites together (through storage and sorting) and thus decreasing the amount of draw calls.

Should be used especially for rendering tilemaps.

SPRITE ANIMATION RENDERING

Used by sprites that will need to be animated. The texture could be composed of multiple sub-rects that will be scrolled through as the rendering is done.

GLSL SHADING EFFECTS

Utilizing the GLSL high level shading language, we can manipulate each vertex / plane to get the desired effect, such as

+ Glow
+ Haze
+ Additive colour
+ Blur
+ Shock/waves

RESOURCE MANAGEMENT (INTERNAL & EXTERNAL)

ASSET RESOURCE MANAGEMENT

This will be used to load, cache, destroy and free our assets. This will decrease memory usage (storing only 1 copy of each resource) and increase performance (referencing the already cached resource instead of making a new one on the fly).

ASSET STORAGE

We need a way to organize all of our assets in a neat and easy to navigate folder structure and possibly have them be in a custom made ZIP-format resource file.

AUDIO

SOUND MANAGER

A static class that will manage all of the audio resources. It will provide easy ways to play sounds and music appropriately. Features: 2D Spatialization which can take mono sounds and place them in the stereo image at a point relative to the observer. Pre-loads short and/or common sounds into memory for instant access. Streams music from the disk. Performs automatic resource clean-up.

SOUND

By Lewis Scott

This section encompasses the aural elements of the game. This includes sound effects (collisions, events), music (menu, lobby, in-game) and entity emission (footsteps, attack sounds, death sounds, idle sounds, etc.).

The music will follow the 'tribal' theme of the game, with soft percussion including timbales, djembe, congas and bongos. It will also include some traditional woodwind and light brass tones. 'Creepier' or more discordant areas will use a more synthetic style with an ominous timbre to create tension.

Sound effects will match in feel with the graphics of the respective entity that they originate from(e.g: the sounds for a particular monster should match the ‘feel’ of the graphics for that monster). Many monsters might be ‘slimy’ or have ‘creepy’ or ‘disgusting’ look to make them obviously the enemy. Many monster sounds may reflect this with ‘moist’ or ‘wet’ sounds. Most monster sounds will be ‘vocalisations’ from the respective monsters.

Event sounds should be satisfying or disappointing, depending on the rectitude of the respective event. Character sounds should be short, simple, and most of all, **not** annoying or grating when played repetitively. These are sounds that will be heard often, so they need to be palatable when heard repeatedly, potentially having variants from which a random choice can be made a playback time.

PLAYER

- Walk/run sounds (including footsteps on different tile types)
- Attack sounds (unarmed and each weapon)
- Death sounds
- Hurt sounds
- Class specific sounds (taunts, abilities, etc.)
- Ghostly noises for ghosts or spectres (theremin)

PLAYER EFFECTS

- Status Effects
    - Bubbling or other effect noises
- Level sound
- Spawn sound

- Victory music
- Buff sounds (shimmering, etc.)

WEAPONS

For each weapon:

- Hit sound
- Miss sound
- Critical sound
- Projectile launch
- Projectile collision(each tile)
- Projectile collision(entity)

MONSTERS

BOSSES

- Main attack charge
- Main attack hit
- Main attack miss
- Secondary attack sounds
- Movement sounds
- Death sounds
- Hurt sounds
- Idle sounds
- Aggro / taunt sounds
- Boss music

MINI-BOSSES

Mini bosses will be placed on the map in small quantities as well as summoned by deities. Each mini-boss will be composed of:

- Movement sounds
- Attack sounds
- Death sound
- Hurt sound
- Idle sounds
- Chase / aggro sounds
- Summon sound(poof or shimmer)

MINIONS

- Movement sounds
- Attack sounds
- Death sounds
- Hurt sounds
- Summon sound(lesser poof or shimmer)

ENVIRONMENT

- Ambient sounds (dripping, creaking, etc.)

DEITIES

- Taunt / catchphrase for selection
- Aspect / ability sounds

INTERACTABLES & OBJECTS

- Rustling leaves
- Pots breaking
- Crates breaking / damage
- Trap setting, activation
- Torches / lamp crackling

PARTICLES

- Sparkles
- Grinding / sparks
- Splatter
- Fire / Crackling

MENU & GUI

- Menu select sound
- Menu movement sound
- Slider / checkbox tick

- Chat send / receive sound

- Splash audio / music
- Menu music

MISCELLANEOUS

- Audio / music for trailer
- Announcements (ex: “PLAYER DEFEATED”, “TIME’S UP!”, “GAME OVER”)

ART

By Rhea Lauzon

This section encompasses the visual elements of the game. This includes sprites, logo work, box art, icons, etc. All assets are organized into categories they belong to.

The art that will be created will be in a **simplistic *¾ perspective pixel art*** with spiritual tribal horror elements matching the chosen theme of ***Dark Fantasy.*** This does not mean **dark colors.**

To elaborate on this concept; players will be given random heads upon game entry with various vibrant colors, patterns, feathers, radical facial expressions, beak faces, etc. Monsters will also follow this theme as well as the rest of the game.

PLAYER

Anything needed for the ‘champion’ that the players control

- Each character will have 4 frames min. in at least 2 directions
- run animations
- class selection
- attribute selection
- 8-16 random heads/masks to be selected on round entry (x2 min. viewing direction)
- attack animations
- death animation
- hurt frame
- on fire animation

PLAYER EFFECTS

- status effects
    - Effects that overcome the sprites such as poison, paralysis, and sleep.
- level up animations
- victory dance
- buffs from deities (ex: health regain)

WEAPONS

Weapons will be a small portion of the game but includes the following. There will be 2-8 weapons in the game; each specific to the class.

- projectiles
- Weapon from different angles

MONSTERS

In the game there are 3 tiers of monsters; bosses, mini-bosses, and minions. Monsters are spawned on map creation and by deities.

BOSSES

Each round will contain ***one*** boss at the center of the map. This boss will be a large creature yielding a high reward. The boss will require the following:

- 4 frames min. in at least 2 directions
- attack frames
- death animation
- hurt frame

MINI-BOSSES

Mini bosses will be placed on the map in small quantities as well as summoned by deities.

- 4 frames min. in at least 2 directions
- attack frames
- death animation
- hurt frame
- summon frames (poof or glow)

MINIONS

Minions will appear on the map most frequently as they are easy little critters.

- 4 frames min. in at least 2 directions
- attack frames
- death animation
- hurt frame
- summon frames (poof or glow)

ENVIRONMENT

Environment composes all things outside of characters and enemies on map. The bulk of this work is the large tileset composing of grass tiles, stone tiles, special tiles, water tiles, and more.

DEITIES

Although deities are not visible on the map there are still some assets needed as following:

- abilities icons
- aspects icons with cool down animations

INTERACTABLES & OBJECTS

This is a general category anything on the map that is physical but non-living. Some of these objects may be breakable as well.

- shrines for each player starting point
- trees
- pots
- crates
- traps
- totems
- lamps & torches with their lighting
- sign posts

PARTICLES

- sparkles
- hit sparks
- blood
- fire sparks
- smoke

MENU & GUI

- Mini map
- mini map markers (ex: green arrow to show where player green is located; must move proportionate to player)
- menu art and icons
  - sliders
  - checkboxes

    - buttons
- chat UI
    - chat box
    - chat buttons
    - chat fields
- HP bar for player with hue changes as it decrement
- HP bar for monsters
- buff and status icons
- player stats
- exp bar
- level display
- scoreboard and timer
- scoreboard icons to show if player is deity, champion, or ghost
- transitions between lobby and game
- splash screen on launch
- victory screen
- enclosing “walls”

MISCELLANEOUS & POLISH

A list of odds and ends that don’t fit in any specific area or will be extras upon completion.

- logo
- game icon for desktop
- box art
- poster art
- trailer art
- announcements at the top of the screen (ex: “PLAYER X DEFEATED”, “TIMES UP!”, “GAME END”)
- tutorial images for controls
- ghost haunting sprite (only visible to ghosts); 2 dimension 3 frame animated

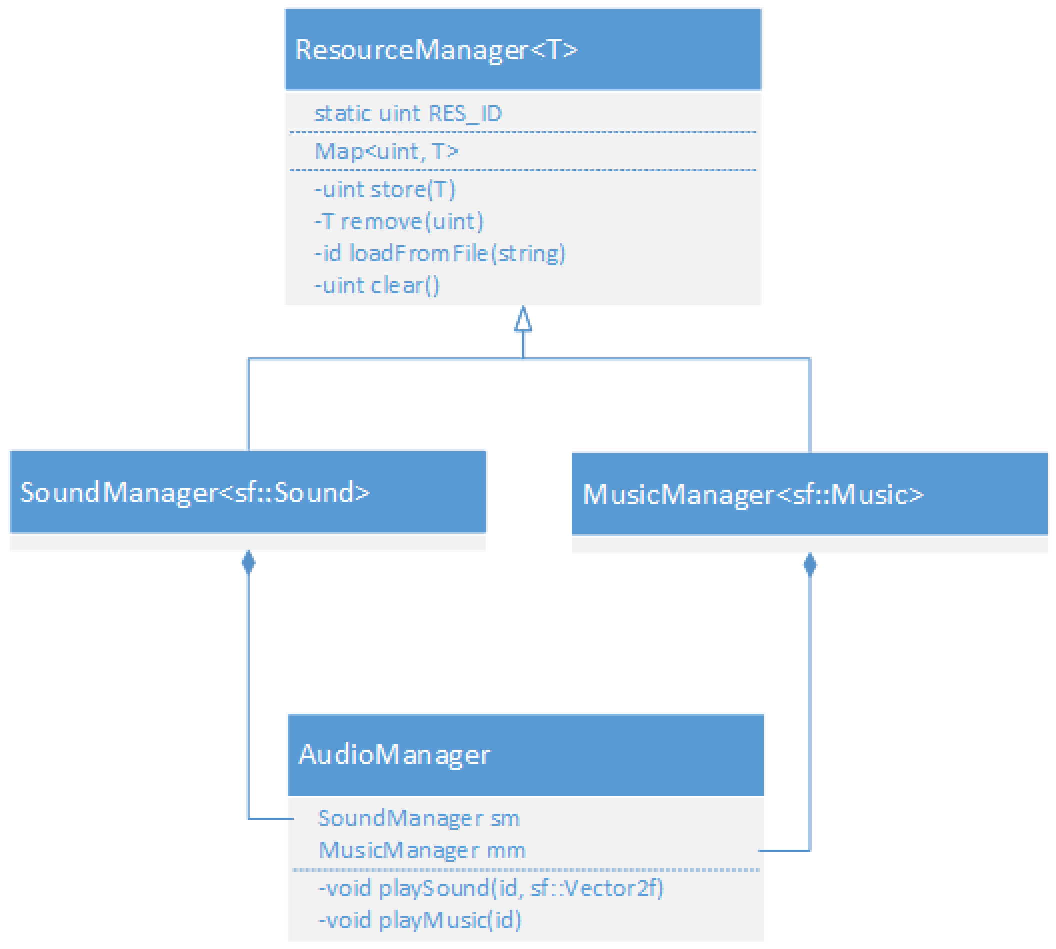
DESIGN WORK

MELVIN LOHO, LEWIS SCOTT, & RHEA LAUZON

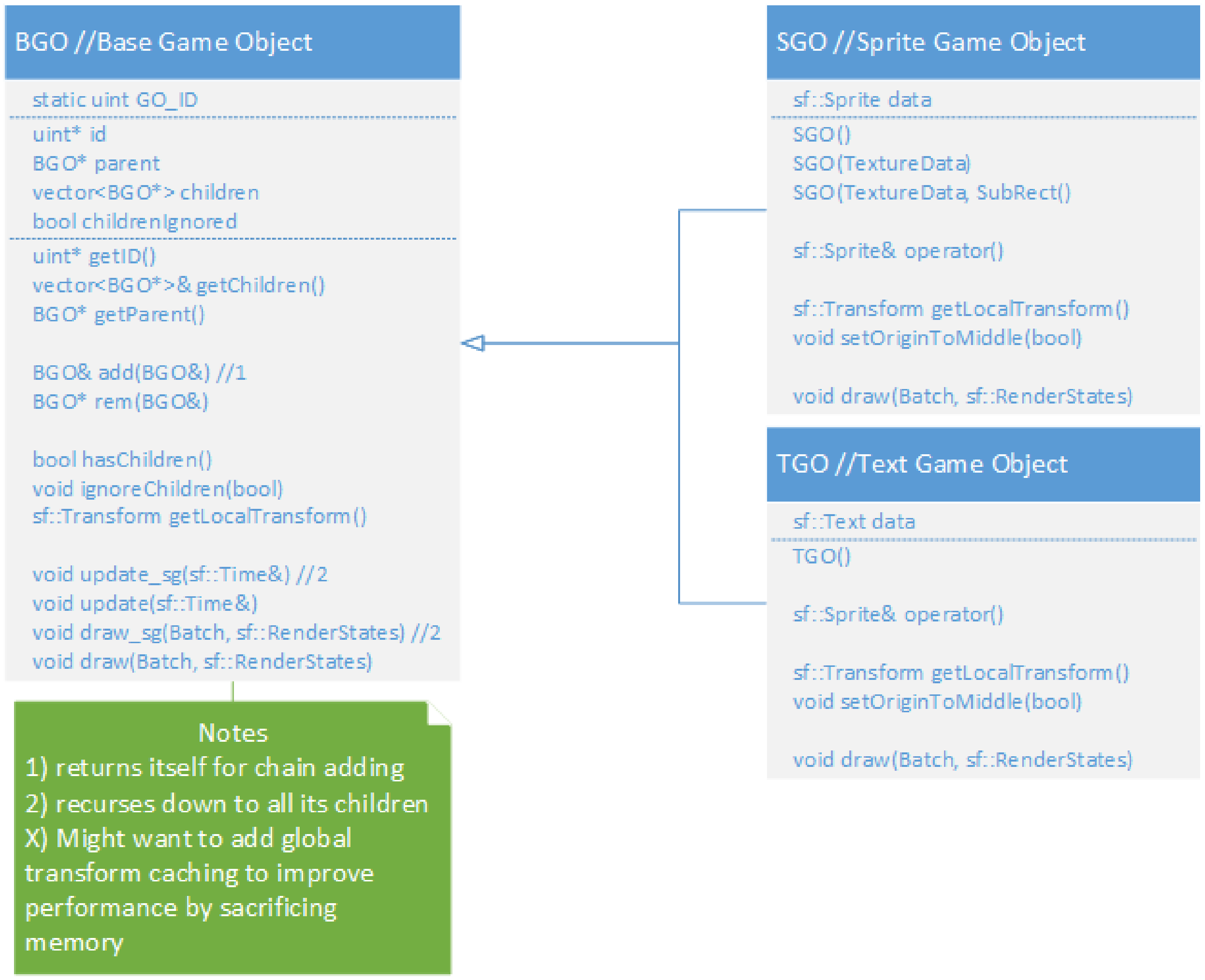
DIAGRAMS

By Melvin Loho & Lewis Scott

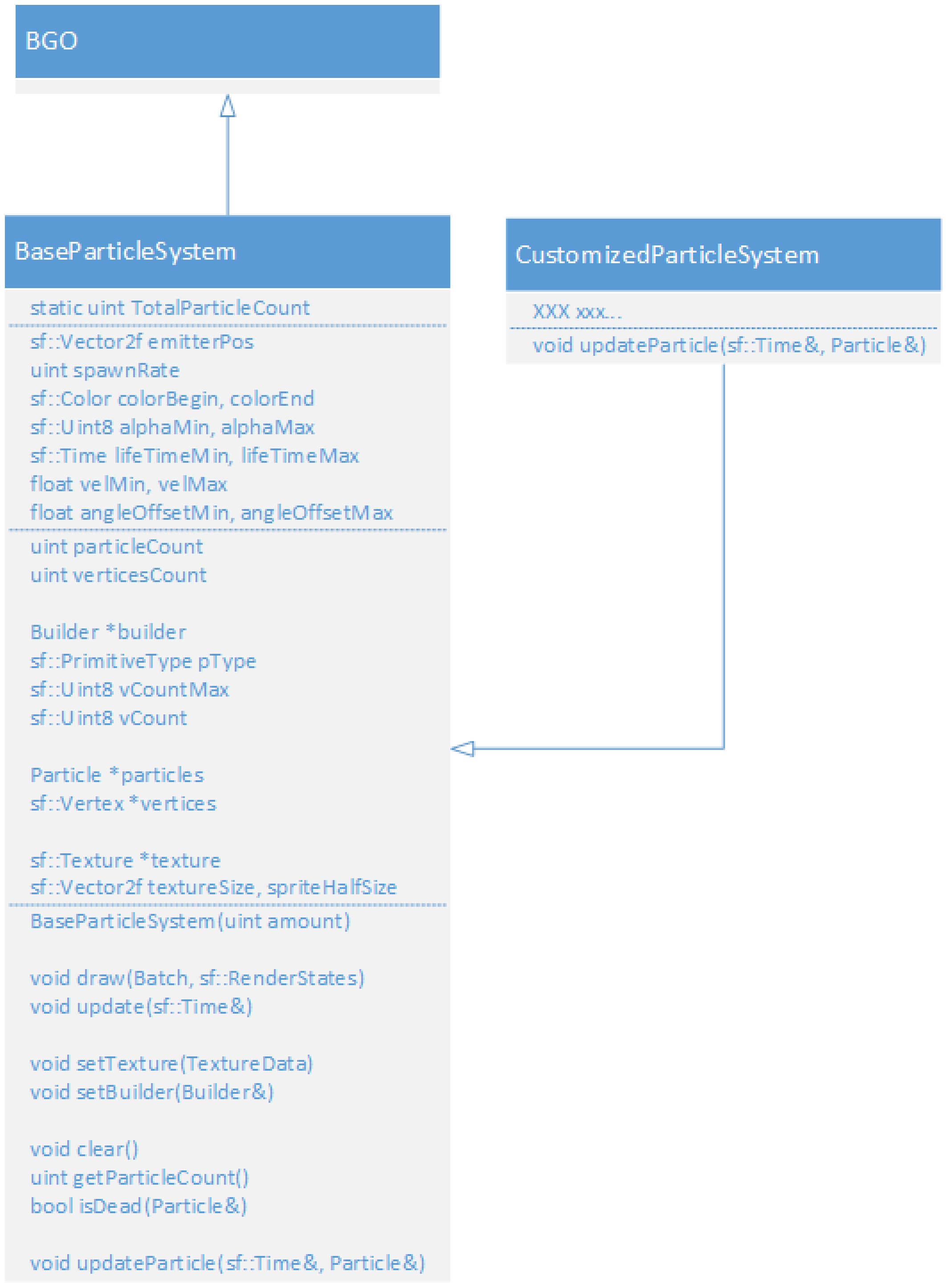
RESOURCE MANAGER



GAME OBJECTS


BGO //Base Game Object
static uint GO_ID
uint* id
BGO* parent
vector<BGO*> children
bool childrenIgnored
uint* getID()
vector<BGO*>& getChildren()
BGO* getParent()
BGO& add(BGO&) //1
BGO* rem(BGO&)
bool hasChildren()
void ignoreChildren(bool)
sf::Transform getLocalTransform()
void update_sg(sf::Time&) //2
void update(sf::Time&)
void draw_sg(Batch, sf::RenderStates) //2
void draw(Batch, sf::RenderStates)
Notes
1) returns itself for chain adding
2) recurses down to all its children
X) Might want to add global transform caching to improve performance by sacrificing memory
SGO //Sprite Game Object
sf::Sprite data
SGO()
SGO(TextureData)
SGO(TextureData, SubRect())
sf::Sprite& operator()
sf::Transform getLocalTransform()
void setOriginToMiddle(bool)
void draw(Batch, sf::RenderStates)
TGO //Text Game Object
sf::Text data
TGO()
sf::Sprite& operator()
sf::Transform getLocalTransform()
void setOriginToMiddle(bool)
void draw(Batch, sf::RenderStates)

PARTICLE SYSTEM


BGO
BaseParticleSystem
static uint TotalParticleCount
sf::Vector2f emitterPos
uint spawnRate
sf::Color colorBegin, colorEnd
sf::Uint8 alphaMin, alphaMax
sf::Time lifeTimeMin, lifeTimeMax
float velMin, velMax
float angleOffsetMin, angleOffsetMax
uint particleCount
uint verticesCount
Builder *builder
sf::PrimitiveType pType
sf::Uint8 vCountMax
sf::Uint8 vCount
Particle *particles
sf::Vertex *vertices
sf::Texture *texture
sf::Vector2f textureSize, spriteHalfSize
BaseParticleSystem(uint amount)
void draw(Batch, sf::RenderStates)
void update(sf::Time&)
void setTexture(TextureData)
void setBuilder(Builder&)
void clear()
uint getParticleCount()
bool isDead(Particle&)
void updateParticle(sf::Time&, Particle&)
CustomizedParticleSystem
XXX xxx...
void updateParticle(sf::Time&, Particle&)

PSEUDOCODE

By Melvin Loho & Lewis Scott

BASE GAME OBJECT

Every game entity, whether it is seen as a sprite, a text object, a combination of both or by something else, needs to be encapsulated in a Game Object. GOs will serve the purpose of helping the things that they hold to exist in the environment correctly (visibility/transformations) and hierarchically. A GO can be added into another GO, thus forming a parent-child bond.

```
BGO
{
    Constructor
    {
        assign itself a new ID;
    }
    Destructor
    {
        destroy its ID;
    }

    getID()
    {
        return its ID;
    }
    getChildren()
    {
        return my list of children;
    }
    getParent()
    {
        return parent;
    }

    add(BGO bgoToAdd)
    {
        if bgoToAdd's id is the same as mine
        {
            throw an error;
        }
        else
        {
            remove bgoToAdd's parent;
            assign myself as bgoToAdd's parent;
            add bgoToAdd to my list of children;
        }
    }
    rem(BGO bgoToRem)
    {
        loop through all of my children
        {
            if the current child matches with bgoToRem
            {
                remove the current child;
                return the removed child;
            }
        }
        return nothing;
    }
    hasChildren()
    {
        if I'm ignoring my children or if I don't have any
        {
            return false;
        }
        else
        {
            return true;
        }
    }
    ignoreChildren(boolean arg)
    {
        I'll ignore my children according to arg;
    }

    getLocalTransform()
    {
        return the Identity Transform;
    }

    update_sg(sf::Time t)
    {
        if hasChildren()
        {
            update all my children by calling its update_sg() method and passing in time t;
        }
    }
    update(sf::Time)
    {
        // empty
    }

    draw_sg(Batch batch, sf::RenderStates parentStates)
    {
        if hasChildren()
        {
            combine my transform matrix with parentStates';

            draw all my children by calling its draw_sg() method and passing in the batch and
combined matrix;
        }
    }
    draw(Batch, sf::RenderStates)
    {
        // empty
    }
}
```

SPRITE TEXT GAME OBJECT

// For the ones that have the same implementation for both SGO and TGO, it'll be called XGO
// "data" refers to either the sprite or text that the GO is holding

```
SGO/TGO
{
       Both Constructors()
       {
              // empty
       }
```

```
        SGO’s Constructor(Texture)
        {
               set the texture of the sprite;
        }
        SGO’s Constructor(Texture, Rect)
        {
               set the texture of the sprite;
               set the texture rect of the sprite;
        }

        Both Destructors()
        {
               // empty
        }

        operator()
        {
               return the data; // sprite or texture
        }

        getLocalTransform()
        {
               return the data's transform;
        }

        setOriginToMiddle(boolean b)
        {
               if b is true
               {
                      get the data's local bounds;
                      divide the bounds by half;
                      set the data's origin to the width and height of the bounds;
               }
               else
               {
                      set the data's origin to 0,0;
               }
        }

        draw(Batch, sf::RenderStates)
        {
               get the batch to draw me;
        }
}
```

BATCH

A rendering technique that increases rendering performance by batching similar sprites together (through storage and sorting) and thus decreasing the amount of draw calls. Should be used especially for rendering tile maps.

```
Batch
{
    Constructor(sf::RenderTarget renderer, maxSprites = 1000)
    {
        initialize variables;
        create vertices holder;
        deactivate self;
    }
    Destructor
    {
        delete/free vertices;
    }

    begin()
    {
        if I'm currently activated
        {
            throw an error;
        }

        reset status variables;
        activate myself;
    }

    end()
    {
        if I'm not activated
        {
            throw an error;
        }

        call flush();
        deactivate myself;
        update status variables;
    }

    prepareDraw(sf::Texture texture)
    {
        if I'm not activated
        {
            throw an error;
        }

        if the texture is not the one I'm currently batching with
        {
            call flush();
            assign my texture to the new one;
        }
        else if I've reached my batching limits
        {
            call flush();
        }

        return the first index for the current batching process;
    }

    flush()
    {
        return if there is nothing to flush;
        update status variables;
        get my renderer to draw my batch;
        increment drawcalls variable;
        reset object count;
    }

    // might change it so my renderstates get combined with the one passed in through this
method
    draw(BGO bgo, boolean scenegraph = false, sf::RenderStates states = Default)
    {
        if we're drawing bgo and its children
            call bgo's draw_sg() method and passing in myself and my renderstates;
        else
            call bgo's draw() method and passing in myself and my renderstates;
    }
    draw(SGO sgo, sf::RenderStates states = Default)
    {
        combine sgo's transform matrix with states';
        calculate and store transformed vertices positions;
        create the appropriate vertices;
        send sgo's texture and the newly made vertices to the last draw call;
    }
    draw(TGO tgo, sf::RenderStates states = Default)
    {
        if I'm not activated
        {
            throw an error;
        }
        call flush();
        get my renderer to draw this tgo with the states;
        increment drawcalls variable;
    }
    draw(sf::Texture texture, sf::Vertex firstVertex)
    {
        call prepareDraw() with the texture and store its return value;
        use the return value to point to the first vertex we can use;
        assign each vertices I'm holding with the vertices passed into this method;
    }
}
```

BASE PARTICLE SYSTEM

Particle systems are containers in which particles are made, manipulated, built, and rendered. They should be extended for uses like making fireballs or smoke emitters.

A particle system holds many variables to control its particles with:

+ lifetime
+ colour over life
+ size over life
+ emitter position
+ gravity
+ texturing
+ particle rotation
+ particle velocity
+ collisions

```
BaseParticleSystem
{
    Constructor(amount)
    {
        initialize my own public variables;
        assign builder;
        create particles and vertices holder;
    }
    Destructor()
    {
        delete/free particles and vertices;
    }

    draw(Batch, sf::RenderStates states)
    {
        if I have a texture
        {
            if I'm just drawing points
                assign states' texture to nothing;
            else
                assign states' texture to my texture;

            increment the batch's drawcalls;

            use batch's renderer to draw myself;
        }
    }

    update(sf::Time)
    {
        spawn and update particles till you can't spawn anymore;

        update remaining particles;

        collide all particles;

        build vertices;

        update status; // total particle count
    }

    setTexture(TextureData newtexture)
    {
        assign my texture to newtexture;
        update my texture size variables;
    }

    setBuilder(Builder newbuilder)
    {
        if the newbuilder's count is bigger than the count I can handle
            throw an error;
        set my builder to the new one;
        update my builder variables;
    }

    addCollisionRect(sf::FloatRect rect)
    {
        add rect to my list of collision rects
    }

    clearCollisionRects()
    {
        empty my list of collision rects
    }

    void detectCollisions(Particle& p)
    {
        for each collision rectangle
        if the vertex is in the rectangle
        collide particle p
    }

    void collide(Particle& p, sf::FloatRect rect)
    {
        //empty
    }
```

```
    clear()
    {
        set all of my particles' lifetime to 0;
    }

    getParticleCount()
    {
        return vertices count / individual vertex count;
    }

    isDead(Particle p)
    {
        if particle p's lifetime is 0 or less
            return true;
        else
            return false;
    }

    updateParticle(Particle, sf::Time)
    {
        // empty
    }

    collideParticle(Particle, sf::FloatRect)
    {
        // empty
    }
}

// the particle builder interface

ParticleBuilder
{
    build(ParticleSystem, particleCount, Particle, Vertices, drawCount)
    {
        // empty
    }

    getType()
    {
        // empty
    }

    getCount()
    {
        // empty
    }
}
```

RESOURCE MANAGER

This will be used to load, cache, destroy and free our assets. This will decrease memory usage (storing only 1 copy of each resource) and increase performance (referencing the already cached resource instead of making a new one on the fly).

```
// ID's could be done the same way I'm doing it for GO's.. maybe...
// (uint*)
// If the above is done, then ID's are destroyable / need to be destroyed once it's no longer
in use (see remove())
ResMgr<T>
{
    load(string path)
    {
        // empty
    }

    store(T)
    {
        create new ID;
        put the new ID and the value T into the resource map;
        return ID;
    }

    store(T, ID)
    {
        if there is already a pair with that ID
        {
            call remove() on that ID;
            put the ID and the value T into the resource map;
        }
    }

    get(ID key)
    {
        search resource map for key;
        if found
        {
            return value;
        }
        else
        {
            return nothing;
        }
    }

    remove(ID key)
    {
        call get() with the key;

        if something's found
        {
            delete it from the resource map;
            destroy the key;
            return the object found;
        }
        else
        {
            return nothing;
        }
    }

    clear()
    {
        get count of resources;
        clear resource map;
        return count;
    }
}

// template specifications:

SndBuffMgr<sf::SoundBuffer>
```

```
{
    load(string path)
    {
        create new sf::SoundBuffer;
        call sf::SoundBuffer's loadFromFile() with the path;
        return the sf::SoundBuffer;
    }
}

SndMgr<sf::Music>
{
    load(string path)
    {
        create new sf::Music;
        call sf::Music's openFromFile() with the path; // <!
        return the sf::Music;
    }
}
```

RESOURCE MANAGER: AUDIO

A static class that will manage all of the audio resources. It will provide easy ways to play sounds and music appropriately. Features: 2D spatialization which can take mono sounds and place them in the stereo image at a point relative to the observer. Pre-loads short and/or common sounds into memory for instant access. Streams music from the disk. Performs automatic resource clean-up.

```
// besides the Music and SoundBuffer ResMgrs that AudioMgr has, it also has a Sound ResMgr
which will link the buffer and the sound objects together
AudioMgr
{
    loadSound(string path)
    {
        call load() on the SoundBuffer ResMgr with the path;
        call store() on the SoundBuffer with the value from above;
        use the return value from above to store a new Sound with the same ID;

        if everything was successful
            return true;
        else
            return false;
    }

    loadMusic(string path)
    {
        call load() on the SoundBuffer ResMgr with the path;
        call store() on the SoundBuffer with the value from above;

        if everything was successful
            return true;
        else
            return false;
    }

    removeSound(ID key)
    {
        make copy of key;
        call remove() on the sound manager with the key;
        call remove() on the sound buffer manager with the key;
    }

    removeMusic(ID key)
    {
        call remove() on the music manager with the key;
    }

    playSound(ID key, sf::Vector2f position)
    {
        find the sound with the ID;

        if not found
            return;

        set position of sound;
        play sound;
    }

    playMusic(ID)
    {
        find the music with the ID;

        if not found
            return;

        play music;
    }
}
```

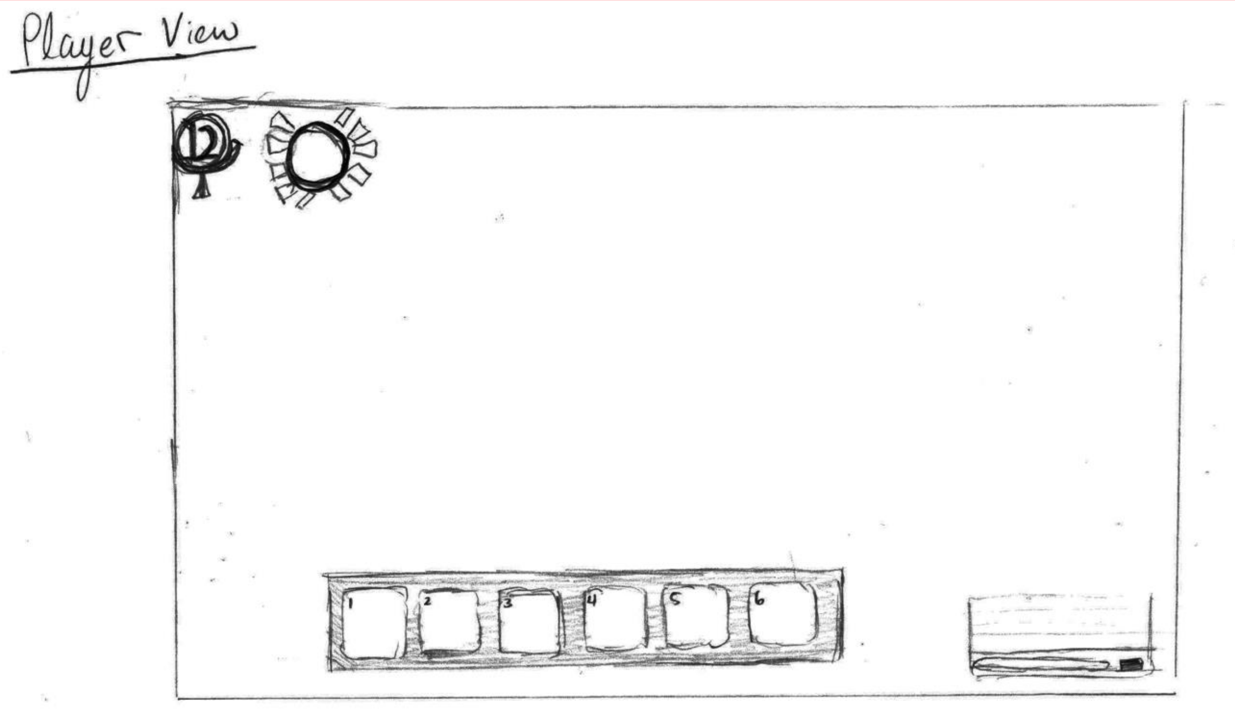
SHADERS

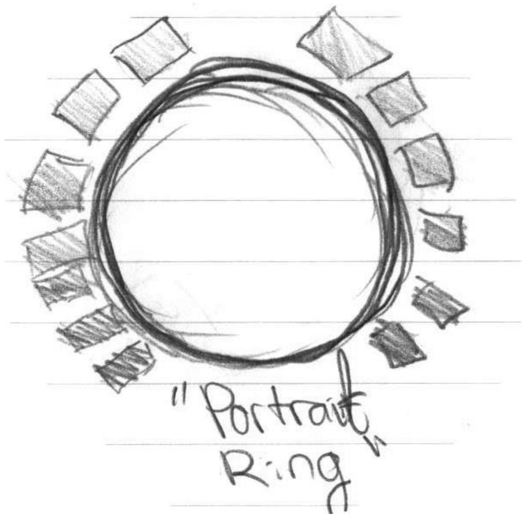
```
blur
{
        calculate the vertical and horizontal offset
        calculate the blur kernel based on input parameter

        add each texture location in the kernel to the pixel color
        multiply each texture location by it's kernel mutator

        set the fragment color based on pixel (divide by kernel size)
}

huesat
{
        set the pixel color to the corresponding texture location
        convert the RGB of the pixel to HSV and apply it to fragHSV

        add hue input parameter to the hue subset of fragHSV

        convert fragHSV to RGB format and apply it to pixel
        add brightness parameter to each subset of pixel

        clamp each subset of pixel
        apply contrast parameter to each subset of pixel

        set fragcolour to pixel
}

lighting
{
        set basedistance to the fragment location
        invert the y element of basedistance //to correct for SFML inverted y-axis

        set distance to origin parameter - basedistance
        set distance to the length of the line represented by distance

        set attenuation to the reciprical of (attenuation parameter * distance) * 2
        set frag colour elements R,G and B to attenuation, A to 1;
        set frag colour to frag colour * lightcolour parameter
}

pound
{
        set vertex to vertex * modelviewmatrix
        set offset to vertex - poundposition parameter
        set length to the length of the line represented by offset

        if length is less than poundfinalradius parameter
                set pushdistance to poundinitialradius + length / poundtotalradius *
(poundtotalradius - poundinitialradius)
                set vertex to poundposition parameter + normalized offset * pushdistance

        if length is equal to poundfinalradius parameter
                set pushdistance to poundinitialradius + length / poundinitialradius squared
                set vertex to poundposition parameter + normalized offset * pushdistance

        if vertex x element is equal to poundposition parameter x element
                square vertex x element
        if vertex y element is equal to poundposition paremeter y element
                square vertex y element

        set position to vertex * projection matrix
        set texture coordinate to texture martrix * projection texture coord matrix
        set the vertex front colour to the vertex colour
}

wave
{
        set vertex x element to cosine of (vertex y element * phase constant + phase parameter
* x cos constant)
        multiply vertex x element by amplitude parameter
        add to vertex x element: sine of (vertex y element * phase constant + phase paremeter *
x sin constant)
        multiply vertex x element by amplitude parameter

        set vertex y element to sine of (vertex y element * phase constant + phase parameter *
y sin constant)
        multiply vertex y element by amplitude parameter
        add to vertex y element: cosine of (vertex y element * phase constant + phase paremeter
* y cos constant)
        multiply vertex y element by amplitude parameter

        set poisition to model view projection matrix * vertex position
        set texture coordinate to texture martix * projection texture coordinate matrix
        set the vertex front colour to the vertex colour
}

colourize
{
        if multiply parameter is true
                multiply fragment colour by colour parameter
        if multiply parameter is false
                add colour parameter to fragment colour

        subtract offset parameter from fragment colour
}
```

UI MOCKS

By Rhea Lauzon

PLAYER (VESSEL) VIEW



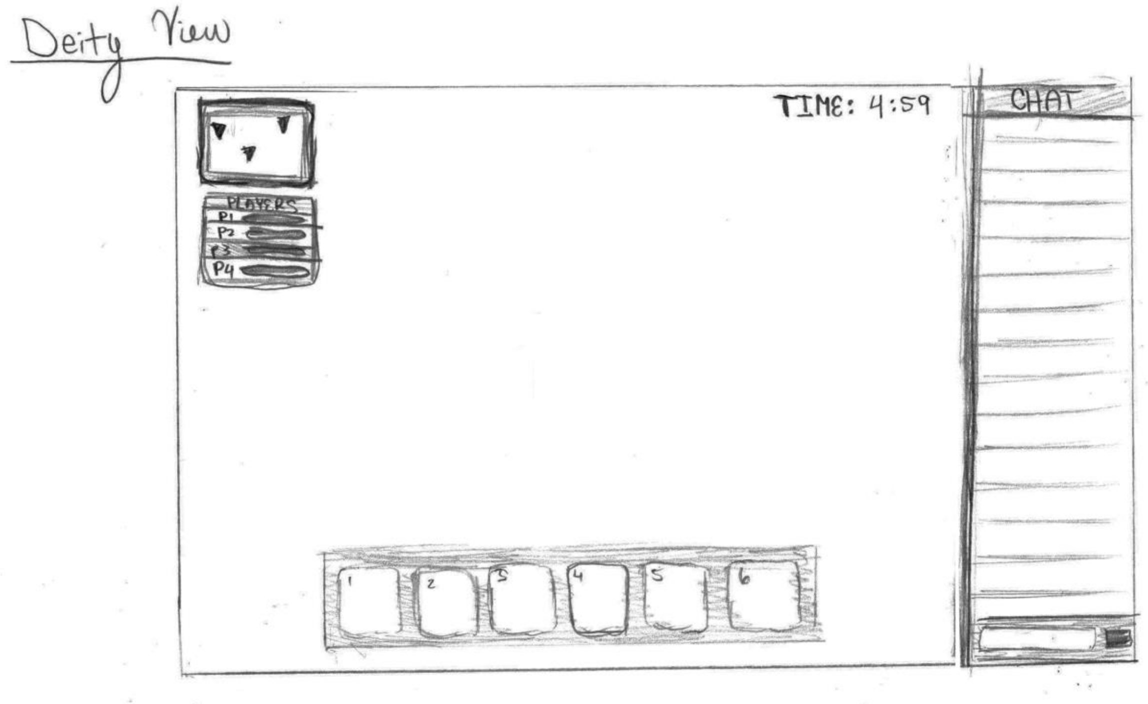


Includes:

1. Health bar surrounding an up-scaled icon of the player's mask
2. level indicator
3. skill bar
4. transparent chat box in bottom left that fades out when inactive

- The experience bar will not be visible to the player except when a kill is made. It will appear as a simple golden bar above the players head for a brief period of time.
- No timer is displayed to the player (at this point)
- Mini map is not visible
- Scores will *only* be visible at the end of each round/match in the form of a scoreboard
- stats and weapon info are not available to the player (not necessary; only one weapon per class in game and stats are not chosen by the player)
- status effects will be visible on the sprite itself therefore no indicators GUI-wise

PLAYER (DEITY) VIEW



Includes:
1. mini map with player markings
2. player selection box with basic HP bar for each player; clicking on a player will bring the deity to that player's view
3. timer counting down
4. chat box
5. skill bar

SPECTATOR (GHOST) VIEW

This view is identical to the deity view minus the skill bar. Upon a vessel's death, the chat box will grow upwards and become present. The skill bar will vanish and the HP / level indicators will be swapped for the mini map.

ARTISTIC VISION

RHEA LAUZON

ARTISTIC VISION

This document is to serve as insight into the artistic and creative visions of the project. This includes a great deal of ***background lore*** to give the developers a mental image of the story to be told. In order to make a game more immersive, it is important to tell a story visually without words. This should be used as a guide to inspire and fuel the story telling objective.

TERMINOLOGY

Soul - Basic being in the realm. Can be a vessel, deity, or a basic soul.

Vessel - A soul that fights directly in the arena. They have a physical form.

Deity - A soul that fights in the arena by aiding or hindering vessels. They do not have a physical form.

Gatekeepers - The general term for all monsters present.

Arbiter - Omnipresent force and overlord of the realm. Also considered the "big boss".

Guardians - The henchmen of the Arbiter.

The Lost - Small creatures that were once souls.

THE WORLD

The game begins with the ***vessels*** being placed on a mystical island that does not exist in our current world. This island is in a realm between life and death; a crossroad realm where spirits of the dead come in order to be judged. Surrounding the island entirely is dark and stormy water, isolating it and providing a sense that this island is potentially the only land.

The island itself is a very flat environment. Rigid Cliffside drops off into the raging sea all around the island with moss and grass growing upon the cliffs. This grassland extends a decent amount into the island on all sides. It is the most tranquil part of the island and is inhabited by various flora and fauna. Not all of them are necessarily friendly.

As you travel inward into the island the grassland slowly dies out and soon turns to gray stone land.   The lighting becomes dimmer. There is still small patches of life existing but the further inland traveled, the more it becomes apparently that there is no life forms living in the most inland, or central region of the island. Instead, ruins of ancient monuments, fallen pillars and other remnants lay upon the stone.

At the very most center of the map resides the altar. This altar serves as a connection between the vessels and the ***Arbiter***, the divine force of the realm, and the neutral being whom controls and decides all. The altar and its surrounding region is lit with candles and ambient lighting.

THE RITUAL

When a person has died in the living world their souls passes on to the cross road, also known as the island. For thousands of years these fresh souls have been placed in an arena by the omnipotent Arbiter in order to be judged. If the ritual takes too long the Arbiter begins to get impatient and slowly encloses the arena. The arena will shrink towards the center and force the players toward the Arbiter's domain.

Each of the souls in the arena is fighting in order to be reincarnated back into the living world. They are given a ***Vessel*** in order to participate. If the vessel loses, they will stay in the realm as a basic deity after the ritual ends. This does not necessarily mean they will never return to the living world though. During the ritual the vessels are expected not only to fight each other, but also the environment. Various monsters and obstacles exist. Since the vessels must kill each other in the end, they must get stronger with only what the land gives.

In addition to the vessels in the arena, deities are selected by the Arbiter to be part of the battle as an outsider affecting the battlefield. Even though the old spirits cannot directly interact with Arbiter, they still have the ability to be noticed for their extreme malevolence, strategic play, or care for the vessels in play. This results in the deity being reincarnated and removed from the realm as well. Deities can aid or harm their fellow spirits that are fighting in the arena. Perhaps they knew the sprit in the previous life and wish to save them, or punish them.

THE SOULS

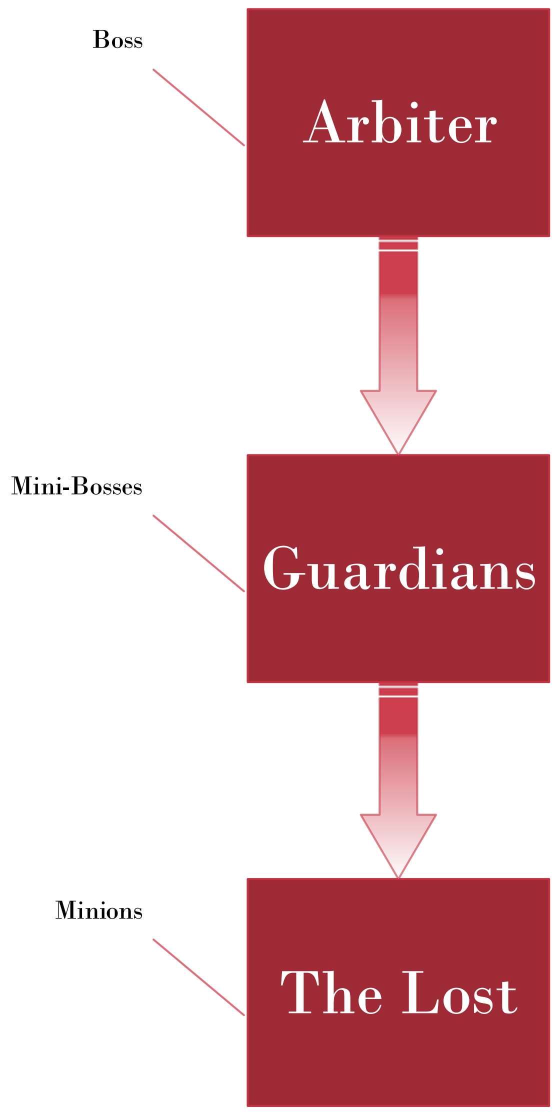
Souls are broken into two categories when they are participating in the ritual: **vessels** and **deities**. Deities have no physical or visual representation and are fluid being. They are only present based off their actions and participation.

On the other hand, vessels *do* have a physical form. The soul has lost its physical body but is given a shadowy humanoid figure in order to do battle. The humanoid body they take possession of is genderless and of neutral build. It serves simply as a vessel for the ritual. However, each champion has one unique characteristic that separates them visually from the rest. Upon the shadowed body's faceless head is a mask, characterised and painted based off the remnants of memories from their living days. In addition to the markings and colors, these masks take various shapes and sizes. When a champion 'dies' while taking place in the ritual their mask shatters and a piece of their memories is lost forever. They then become basic souls that can view the ritual till the end.

THE GATEKEEPERS

The gatekeepers of the realm are of spiritual nature. On the outermost edges of the island the living critters are much more tame and natural feeling. As the gatekeepers reside closer to the Arbiter's grounds they become more abstract and bizarrely shaped. The gatekeepers come in three tiers; ***The Lost,*** ***Guardians,*** and the ***Arbiter***.

Gatekeeper Higharchy



The Lost are the most lesser of gatekeepers and are simple creatures. They are not particularly strong or intelligent. The Lost are souls that once participated in the ritual but performed so poorly that the Arbiter gave them lesser forms. They generally have imperfect shapes and are small in size.

Guardians on the other hand are essentially the pawns of the Arbiter. They can traverse the entire map and vary in appearance. Guardians are slightly larger than a vessel and have a variety in attacks.

THE ARBITER

In addition to small monsters, the vessels may also challenge the **Arbiter** to a duel via the central temple. The Arbiter is the divine being of the realm. It is a neutral being created to pass judgement on the quality of the spirits.

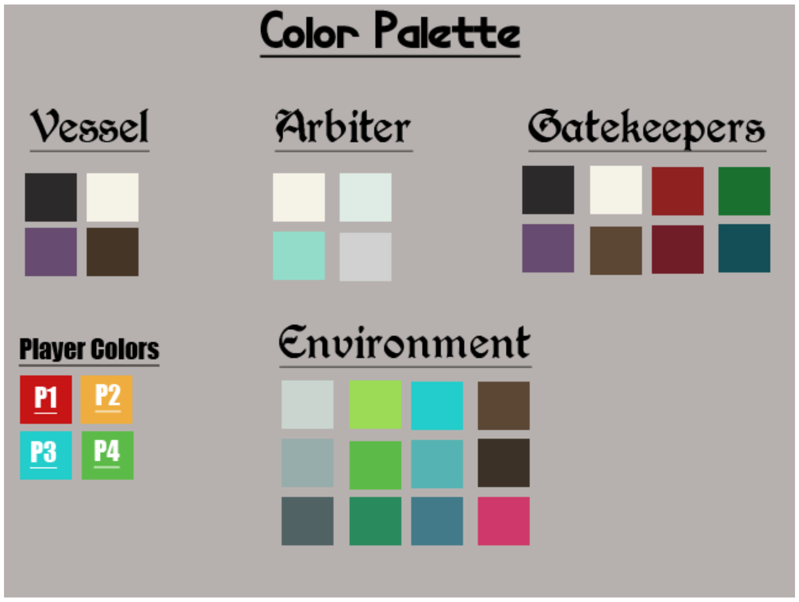
Appearance wise, the Arbiter is based off a white-blue-gray color scheme, contrasting the dark environment. It is also the only other humanoid creature in the arena other than the vessels, but is at least 3 times the size of a champion. The Arbiter wears a mask, but in a different style than the vessels. The masks has simple features and is of one single pale color. It wears long and bulky robes coving its hands and feet giving it an endless feeling.

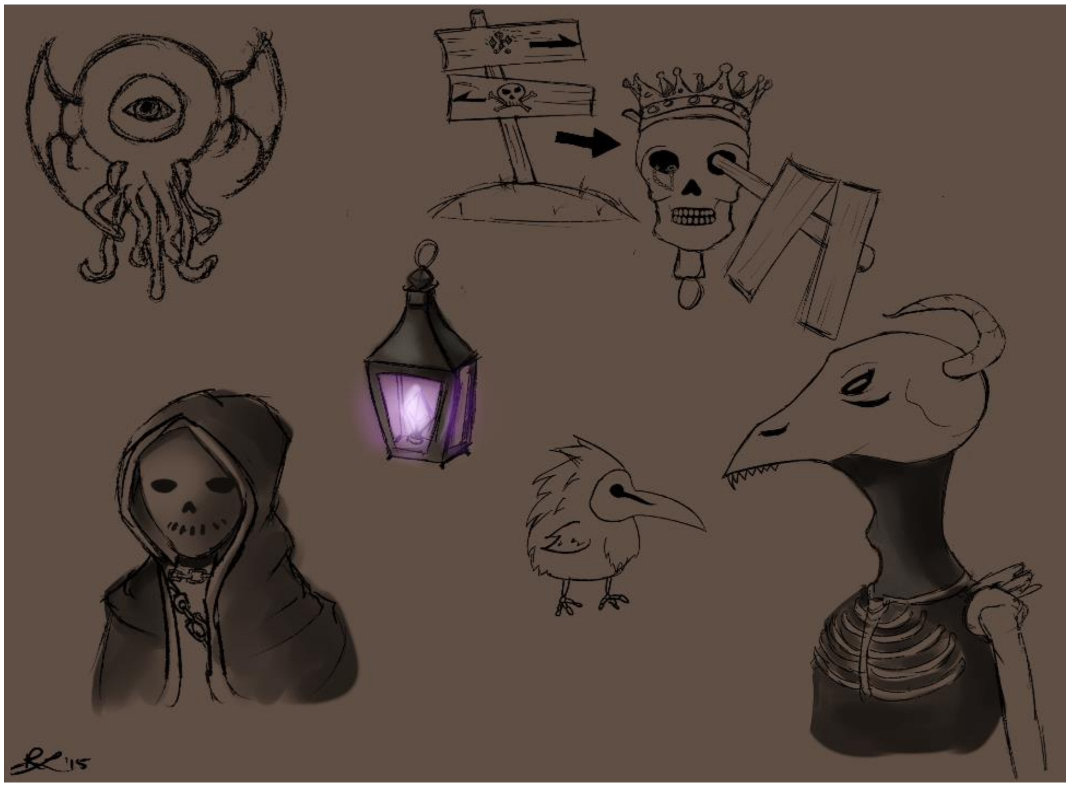
The Arbiter does not appear immediately in the arena. It is either challenged by a champion and thus summoned onto the map or may appear after some time. Its motions are fluid and calm with an eerie edge. Its presence is daunting and commanding.

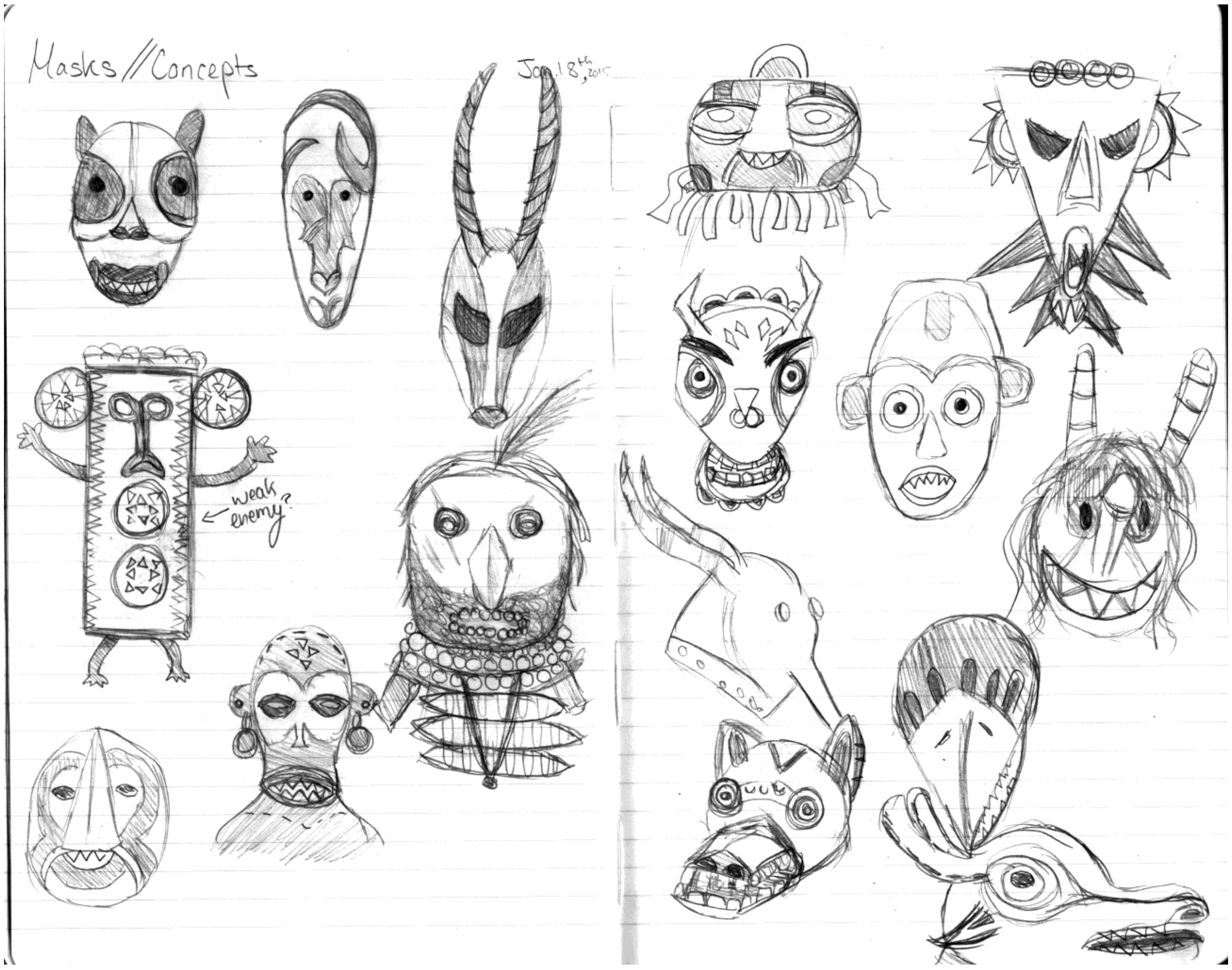
Upon bringing the Arbiter to near-death conditions, the Arbiter will become un-attackable and will bless the champion whom dealt the last blow. A blessed champion appears white instead of the original black and dons a new mask identical in characteristics to the Arbiter's.

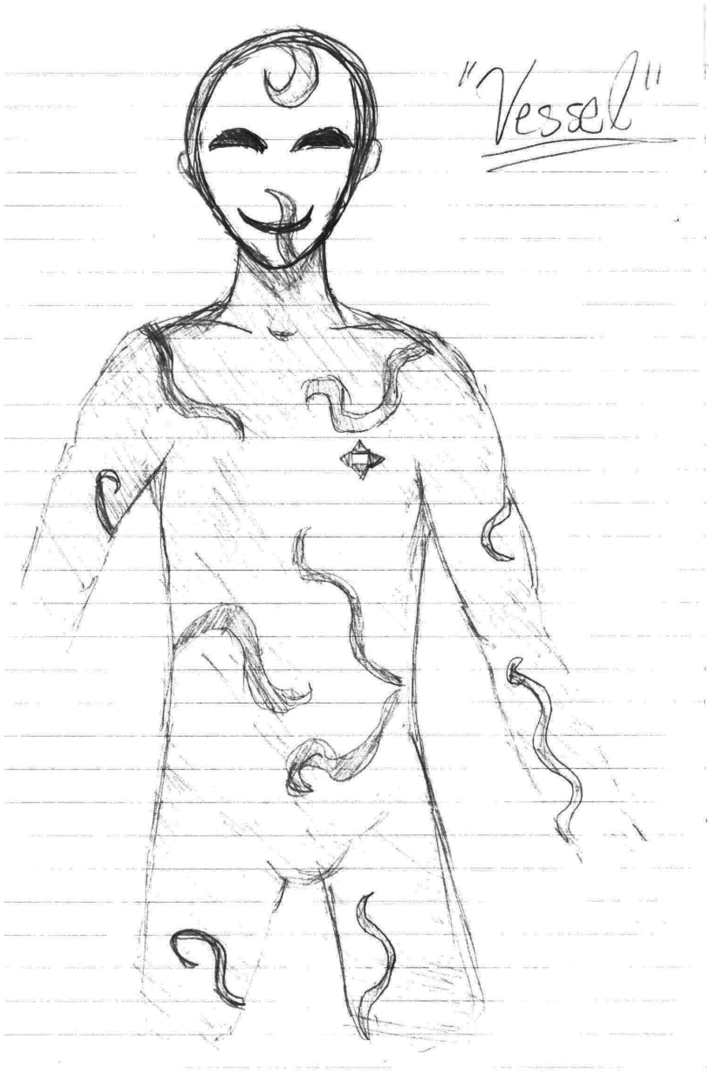
If the Arbiter kills a champion, the champion is remove from the arena as it has lost the challenge. The Arbiter will remain in arena awaiting the next champion. It is unable to leave its temple grounds.

VISUAL CONCEPTS






Masks//Concepts
Jan. 18th, 2015
weak enemy?


"Vessel"