**Design work**

Multimedia team

Contents

[**Overview** 2](#_Toc410757933)

[**Diagrams** 3](#_Toc410757934)

[Resource Manager 3](#_Toc410757935)

[Game Objects 4](#_Toc410757936)

[Particle System 5](#_Toc410757937)

[**Pseudocode** 6](#_Toc410757938)

[Base Game Object 6](#_Toc410757939)

[Sprite Text Game Object 7](#_Toc410757940)

[Batch 8](#_Toc410757941)

[Base Particle System 10](#_Toc410757942)

[Resource Manager 12](#_Toc410757943)

[Resource Manager: AUDIO 14](#_Toc410757944)

[Shaders 16](#_Toc410757945)

[**UI Mocks** 18](#_Toc410757946)

[Player (Vessel) View 18](#_Toc410757947)

[Player (Deity) View 19](#_Toc410757948)

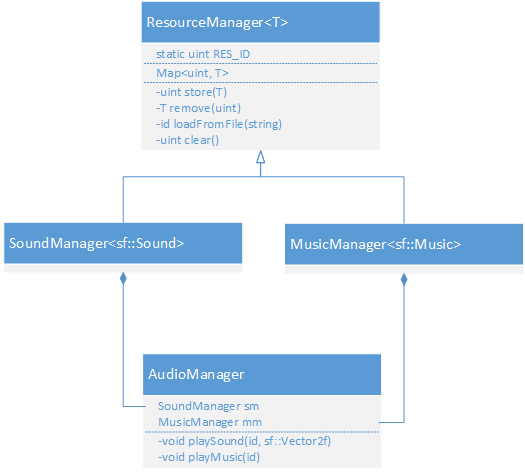
[Spectator (Ghost) View 19](#_Toc410757949)

# **Overview**

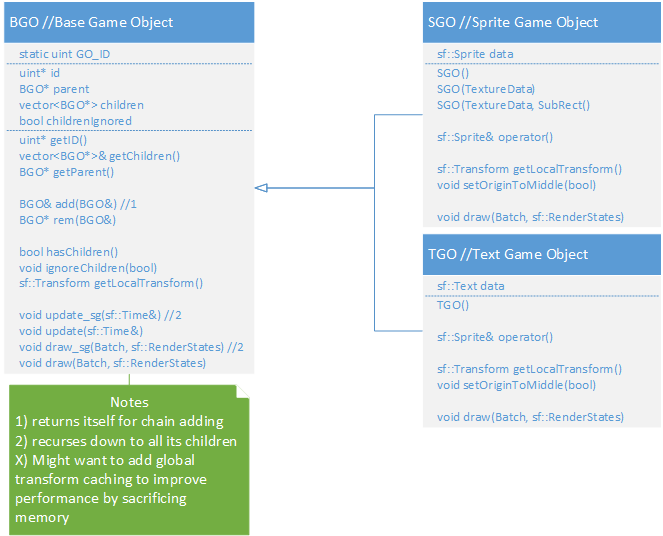
This document contains all flow diagrams, pseudocode, class structures, and any other design related assets created by the multimedia team. These diagrams have been separated from the work breakdown for clarity.

# **Diagrams**

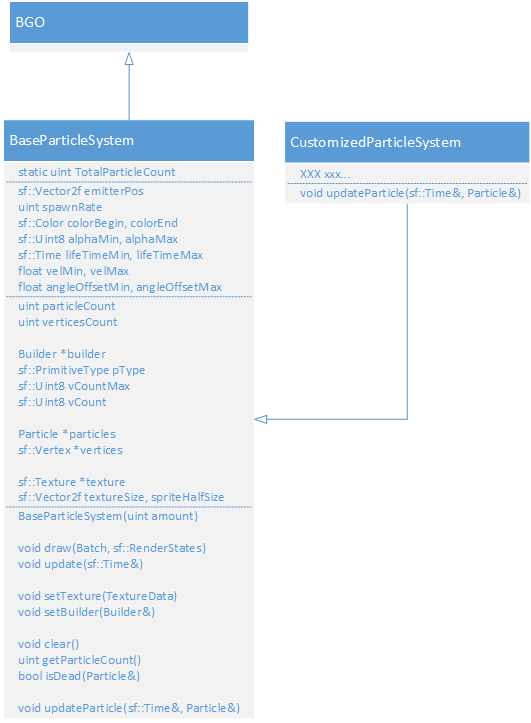
## Resource Manager



## Game Objects



## Particle System



# **Pseudocode**

## 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

}

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

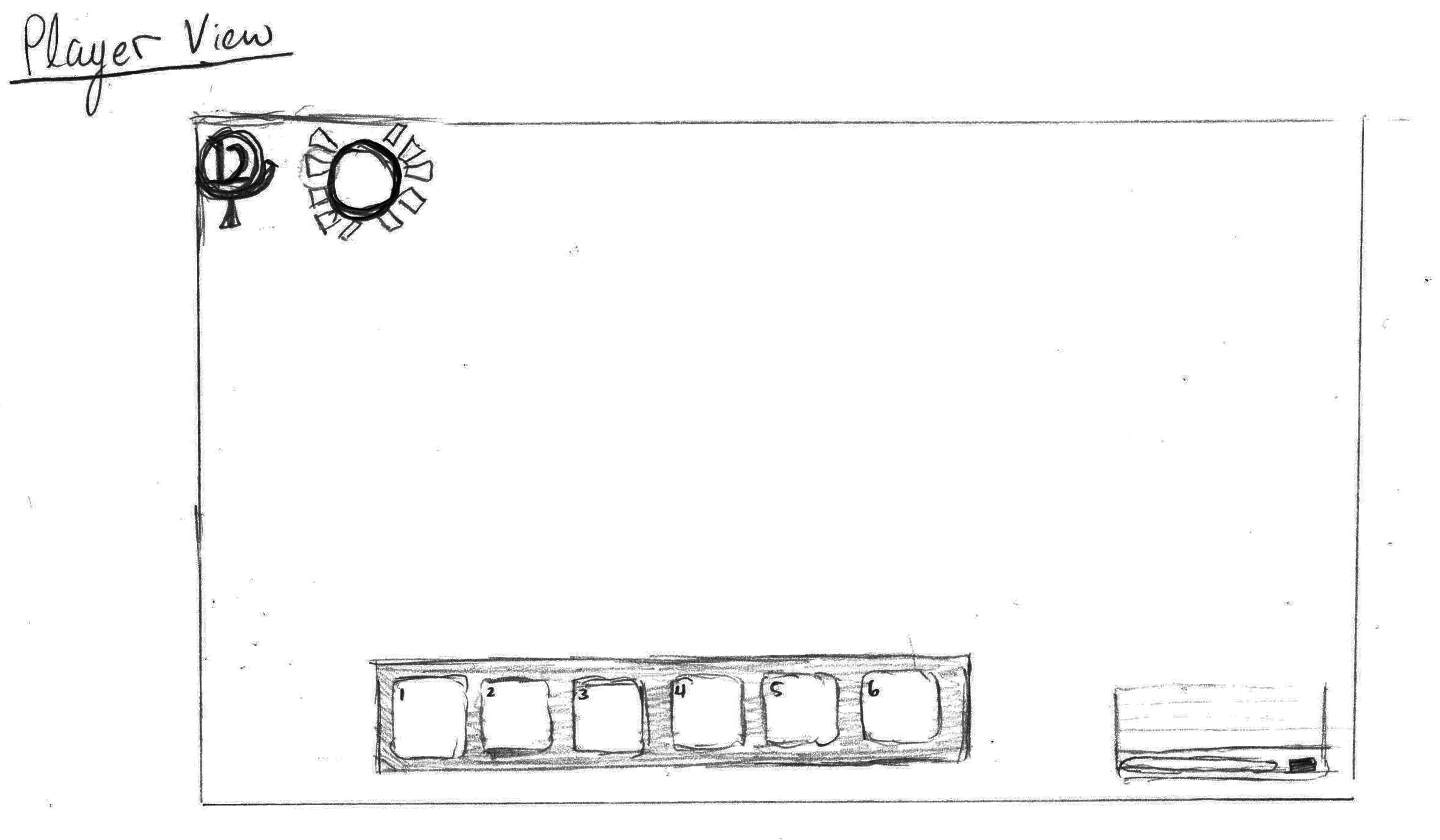
add colour parameter to fragment colour

subtract offset parameter from fragment colour

}

# **UI Mocks**

## Player (Vessel) View



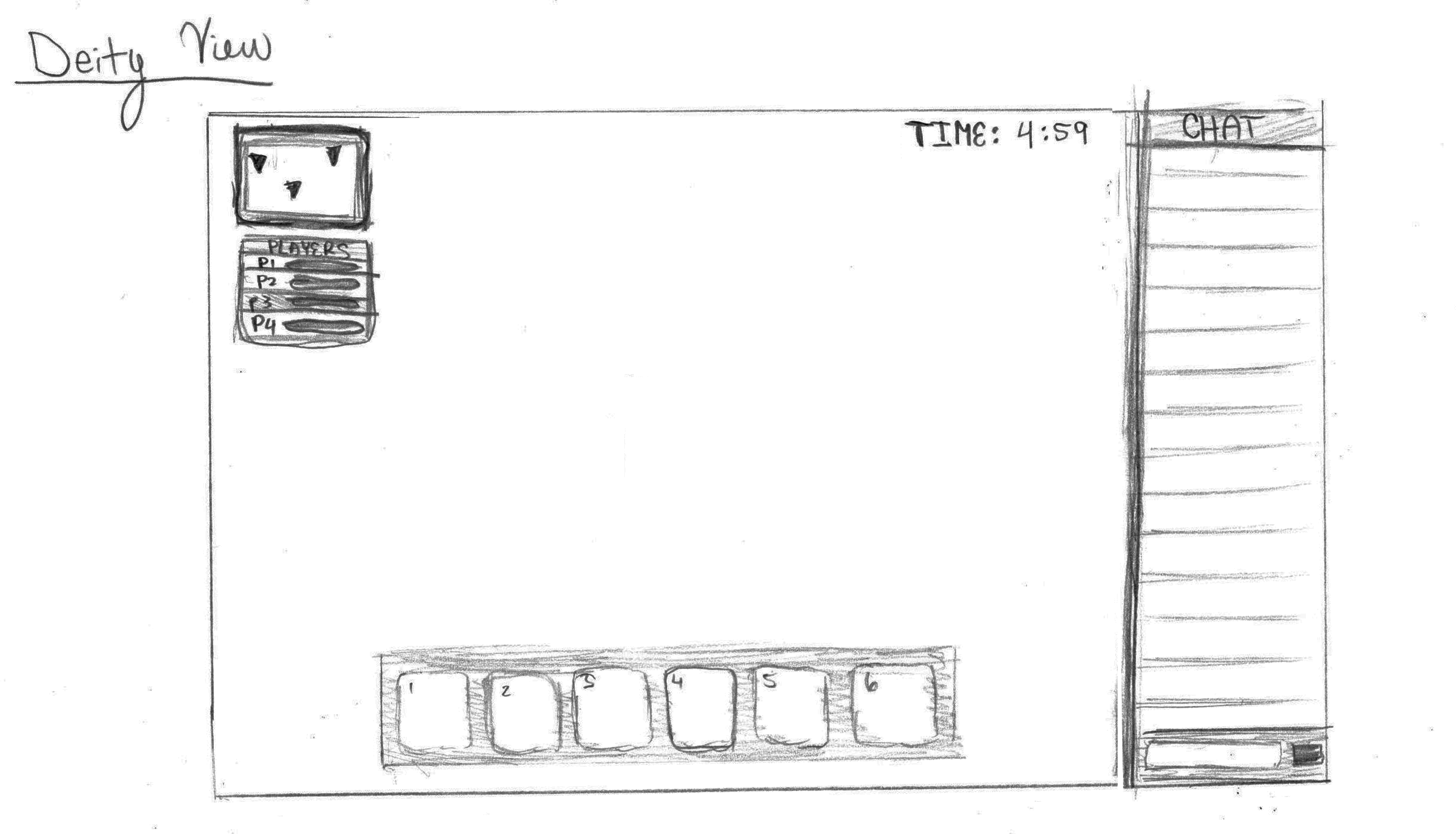


**Includes:**

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.