Design Document Game Engine Team

Document Compiled by Marc Vouve of work from: Marc Vouve, Marc Rafanan and Thomas Tallentire

# AppWindow

The app window is responsible for running the game loop. The game loop processes turns for all of the scenes. Each scene is loaded into the app window on creation and should not have a loop of its own for the sake of synchronisation.





AppWindow

{

Collection of Scenes scene

GameLoop()

{

foreach scene

ProcessEvents

Update

Draw

}

addScene(){}

removeScene(){}

}

# Entity Manager

All Entities that need to be updated each turn will be loaded into the entity manager. The entity manager will persist for an entire world



EntityManager

{

Container entity

Add( Entity e )

{

add Entity to container

}

Remove( Entity e )

{

remove Entity from container

call entity function for onDestroy.

}

TakeTurn()

{

foreach entity

take turn

}

}

# Entity

The entity is the base object within the engine. Other than projectiles, Entity functionality will be handled by the Logic team. Entities do not have to be in the Entity manager, however in order to take a turn in the proper sequence they should be.

The logic behind basic movement is complex so the FSM is specifically used for that.



Entity()

{

Entity(Controller)

{

onCreate();

Assign Controller to this object

}

turn()

{

Foreach( Controller::getEvents() )

{

run event

}

Controller::ClearEvents

}

// This is a bit codey for pseudo-code, but it's done very particular

// this function returns the entity collided with on failure and NULL on success.

Entity move(float x, float y, bool force = false )

{

if( entity is 1 cell in size )

{

Store all cells in the entities range in an set without duplicates

if( !force ) // This argument is passed to check if bounds checking is required

{

Check if any entities are overlapping.

if( entities overlap )

return other entity

}

assign collection of cells to this as it's cells

// remove from current.

Remove Entity From Previous Location

Assign Location and bounding location

}

}

checkCollision( Entity e )

{

if( entity's bounds overlap )

{

return true;

}

}

getCells()

{

returns cellSet

}

onCreate()

{

// logical implementation

}

onDestroy()

{

// logical implementation

}

onUpdate()

{

// logical implementation

}

}

# Controller

The controller is the main “variable actor” in the world. This means that entity commands come from here.

Controller

{

eventQueue;

getEvents()

{

return eventQueue;

}

}

# Player Controller

Simple Input handler, abstracts inputs to allow inputs to be variable

PlayerController extends Controller

{

getEvents()

{

return eventQueue;

}

onMouseButtonDown()

{

}

onKeyPressed()

{

}

... etc general inputs ...

}

# Lobby

The lobby is a persistent object between rounds, and does not get unloaded from world to world. Have people

Lobby

{

playerQueue[]

spectatorQueue[]

int playerLimit // Network or logic team to set

int spectatorLimit // Network or logic team to set

bool joinBlock

// lobby functions

join(player)

{

if(spectatorQueue[].length < spectatorLimit)

Add player to spectatorQueue

else

player cannot join lobby

if(joinable())

move player from spectatorQueue to playerQueue

}

leave(player)

{

Remove player from PlayerQueue

move first player from spectatorQueue to playerQueue

}

joinable()

{

if(playerQueue[].length < playerLimit && !joinBlock)

return true

else

return false

}

changeScene()

{

Do something like starting the game etc.

}

}

# Player

A player holds some basic data about the player, this gets held round to round in the lobby. This should not be confused with an entity

Player

{

playerID

playerDataStructure

}

# Projectile Manager

This object manages projectile creation and deletion in order to conserve memory.

ProjectileManager

{

// action is the skill or type of projectile is it should probably just be a reference to something that

// projectile stores.

// the receiver is a pointer to a cell so if it is tracking it will try to track the cell of the player it's going towards.

// I have NO IDEA if this will end up working in practice but it seems cool.

GetProjectile( Ref action, BGO sender, pointer Cell receiver )

{

if( projectileQueueEmpty )

{

pro = new Projectile( Ref action, sender )

}

else

{

pro = projectileQueue::head

}

entityManager.add(pro)

}

enqueue( Projectile )

{

queue.add( Projectile );

}

onDestroy()

{

delete queue and content.

}

}

Projectile extends entity

{

setSpeed(speed)

{

\_speed = speed

}

onHit()

{

Action::onHit

speed = 0

ProjectileManager enqueue this // If this should be persistant it should be handled in actions onhit, and be converted to some other entity.

entityManager.remove(this)

}

onDestroy()

{

ProjectileManager.enqueue( this );

}

}

// This should be referenced by a projectile, when it is created is unknown, that's up to logic team, I would advice attaching it to a character

// and passing it to the function by reference.

Action <-- needs to be implemented by logic team.

{

onHit()

{

// Logic team implementation, this can be whateven

}

}

# Scene

The basic scene will be created by us to act as a blank scene for the scene loader. Additional scenes will be created by another team. This may be logic or multimedia

Scene

{

OnLoad

{

// Logic implementation

}

unLoad

{

// If no scenes in queue, add blank scene

}

events

{

process events in game

}

update

{

update scene // logic implementation

}

draw

{

// Draws window, multimedia Implementation

}

}

# World

The “world” is a unique instance of the game. It contains a map, an entity manager, and all the entities

// This is a single instance of a game

World

{

Map map

EntityManager manager // Manager for when game loop is going

Array cells[][] // Could be the map, we may want some abstraction though

Container entities // Holds all entities in current game instance

Lobby \_lobby

// World is held in Lobby, but also needs some information from lobby

World (Lobby lobby)

{

\_lobby = lobby;

}

init()

{

foreach cell in cells

cell.init()

map.load()

}

setCell(Cell c, Entitiy e)

{

c.addEntity(e)

e.setCell(c)

}

getEntities()

{

return entities

}

getCells()

{

return cells

}

}