GAME DESIGN AND ENGINEERNG PROTOTYPE DOCUMENTATION

COMP2351 GAME PROTOTYPE MILESTONE

Student No:18010943

Original Game design

Our game is set in the late 1700s and the player will play as James Johnstone Sr, a surgeon from the Worcester infirmary. The objective of the game is to save as many patients as you can before the timer runs out. To save the patient, the player must first workout which illness the patient has by looking at their symptoms. A mechanic in the game to help the player do this is called the handbook. The handbook will allow the player to input symbols into a calculator, which will return the tool that the player must use on the patient in order to treat them. Now that the player knows the tool that they must use, they will walk over to the tool bench, which will prompt them with all the available tools. The player chooses the tool they want to use and then walks over to the patient. Interacting with the patient, while holding the correct tool will start an activity that the player must complete to treat the patient. If they fail, the patient will die, and the player will not gain any score. If the player completes the activity successfully, the player will gain some score. The aim of the game is to end up with the highest score possible before the timer ends.

Link to PowerPoint that describes the game design in further detail

<https://uniworcac-my.sharepoint.com/:p:/r/personal/popp1_19_uni_worc_ac_uk/_layouts/15/Doc.aspx?sourcedoc=%7BFC037744-E204-400E-BF67-71CC84A38CB0%7D&file=Pre-Production%20Milestone%20Presentation.pptx&action=edit&mobileredirect=true>

The Functional requirements for the prototype

MUST HAVE SHOULD HAVE COULD HAVE WILL NOT HAVE

Player

Player movement

Collision with entities

Interact with patients

Interact with tool bench

Player can hold a tool in inventory

AI

Patient has an illness

Randomly choose an illness for the patient when initialised

Health Bar for the Patient

Patient takes damage from activity

Generate a new illness when the patient dies

UI

Main Menu

Handbook UI

Tool bench UI

Symbols for each symptom and body part that links to patient illness

Health bar UI

A score card that pops up when the timer runs out, displaying how well you performed on that level.

Score counter

Game Timer

Scene

Restart game when game timer ends

Treatment items (tools)

Bone saw

Leeches

Trocar

The Gorget

Activities

Quick-time event for the Bone saw

Can have multiple activities active at one time

Activity for the leech behaviour

Add a visual representation for the ‘leech points’ so that the player knows when they are close to finishing the activity

Handbook

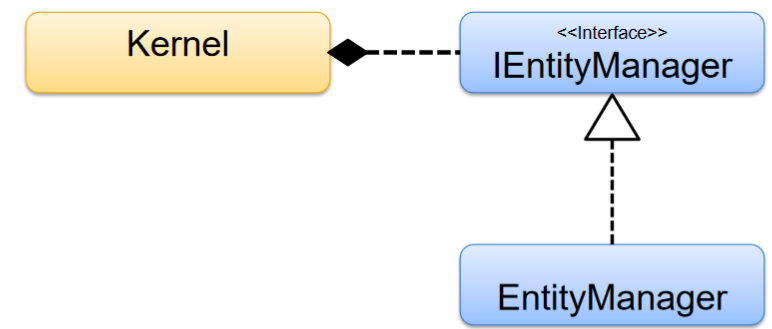
Player can click on symbols to add them to the illness calculator

Player can clear the calculator to input new values

Handbook pauses the game when it is active and un pauses the game when it is not

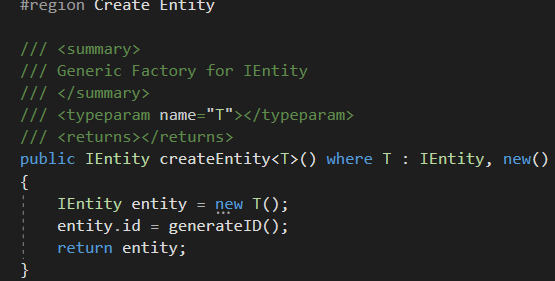
Entity Management

In Game Development, an entity is a component that is needed to incorporate gameplay functionality. Entity management includes the creation and termination (removing from the scene manager) of these components.



This is the class structure that I used for managing my entities in the prototype. The kernel calls method from the IEntityManager interface to create and remove entities from the scene. The EntityManager uses an Abstract Factory method to create and return IEntities back to the kernel.

Abstract Factory for creating an IEntity



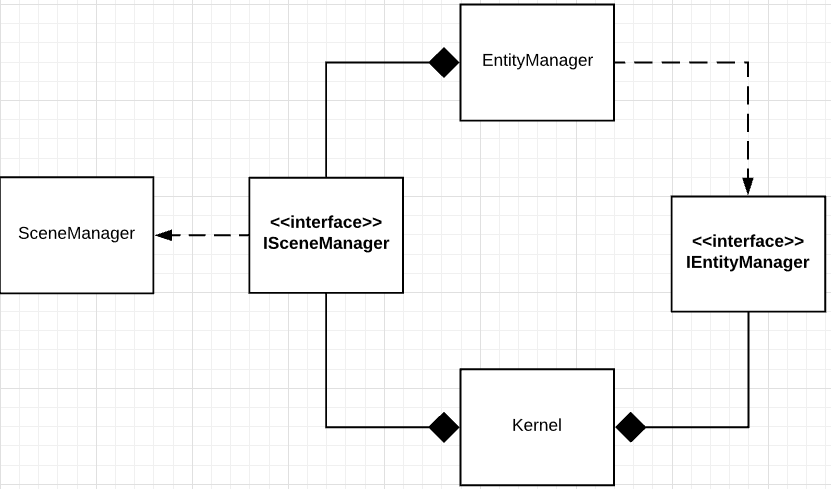
This line of code is taken from the kernel and is used to call upon the EntityManager to create a Button



This line of code is also taken from the kernel, but this is used to remove the player from the scenemanager

Scene Management

In game development, Scene management involves adding and updating the entities on the game scene. In my prototype it also draws the entity onto the scene, using the sprite batch created in Kernel.



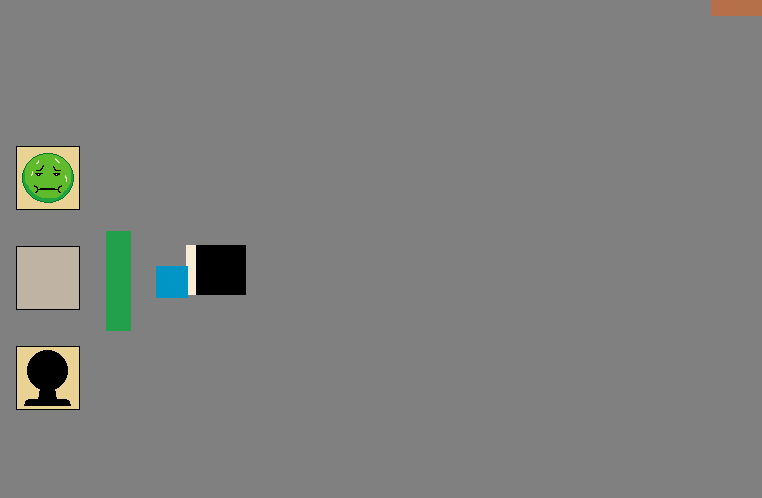
This is the Class structure of the Scene Management architecture. The Kernel uses the ISceneManager to add the entity it just created using the IEntityManager, to the game scene. An example of how I used this in my game prototype is below.





The kernel sets the entities texture, the scene manager draws the entity onto the scene using this texture.

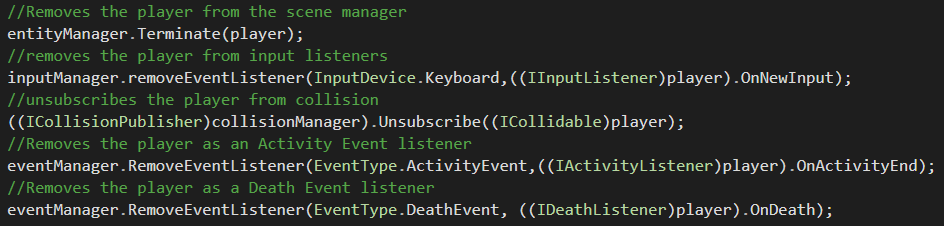




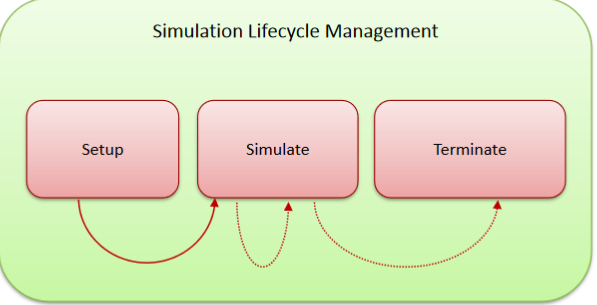
This image shows the entities being drawn onto the scene.

Entity/Simulation Lifecycle Management

Entity Lifecycle Management involves managing the many state of an entity in a game world. The states are called Create, Initialise, Update and Terminate. The main problem I have with lifecycle management is the termination of entities. The reason for this was because before removing the entity from the scene, I had to also remove the reference to the entity that associated object had to the entity. For example, in my prototype I have create a class that manages all events occurring in the game. At the moment, I have kernel subscribing each entity, but what I wanted to do is add a class called game manager, which took care of subscribing each entity as a listener for an event. It would also unsubscribe these entities from the event and remove them from the scene manager. To achieve this, I would use polling and each entity would flag when they want to be removed and the game manager would get this value to see if it true. If it is true, then remove them from the scene, else do nothing.



The above code is for terminating the player by removing it from all associated objects. This code is executed after the gameEnd event is fired, which the Kernel listens for.

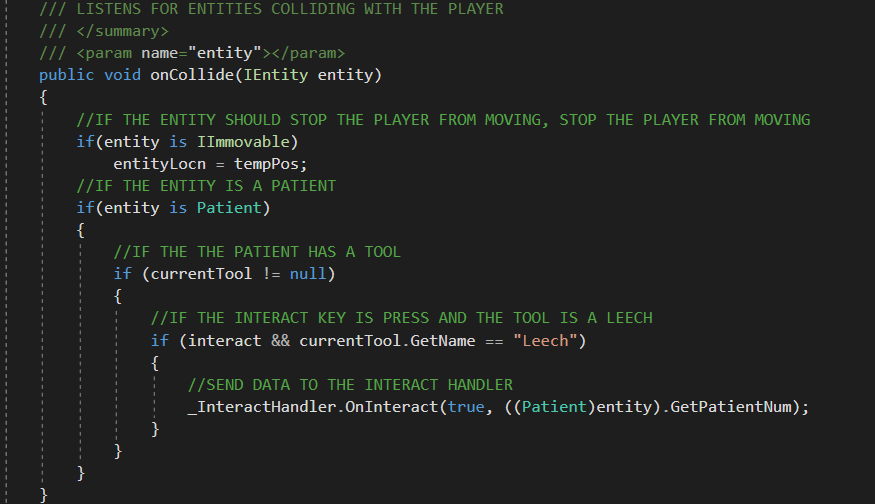


Simulation Lifecycle Management

(found at Game Engineering lesson 09-Simulation Orchestration)

Collision Management

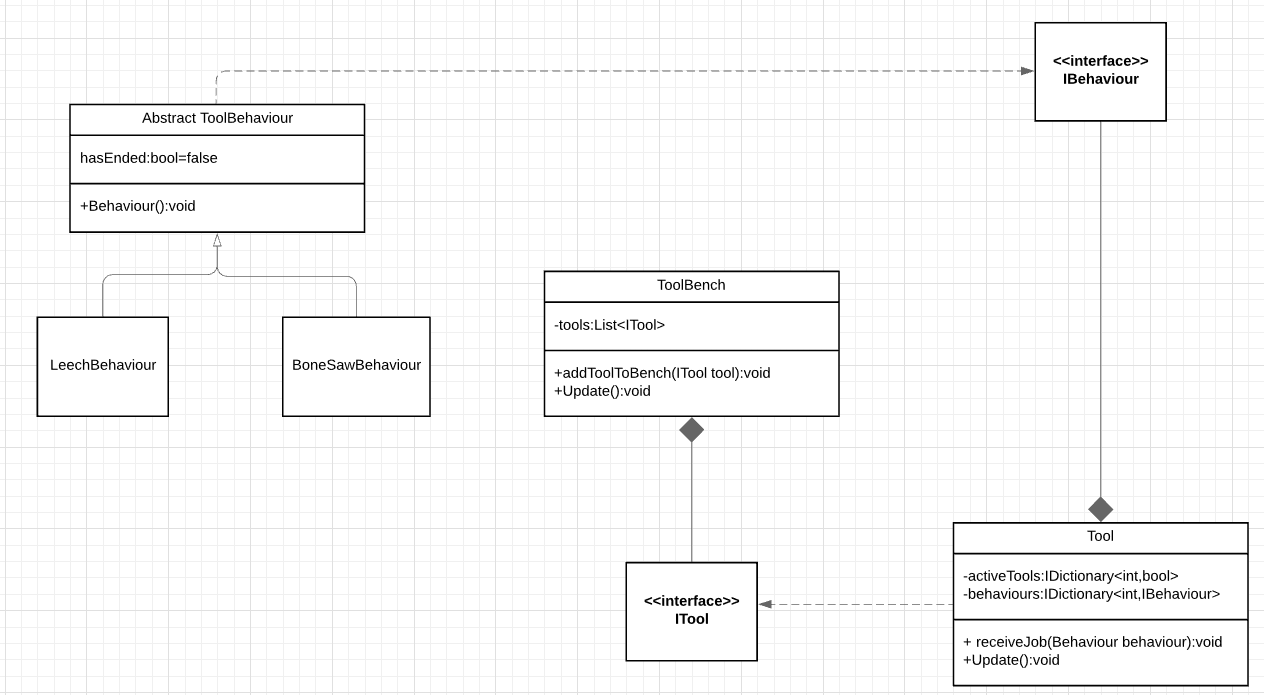
To accomplish collision management in my game prototype I use the polling technique I used in my team’s game engine which involves using the observer design pattern. The observer allows an object to publish its state to other objects and these objects can subscribe to the event to be notified with any changes (Carr, 2009). This can be extremely useful for behaviours that only happen every now and again. There needs to be at least 2 entities involved in this pattern. One which publishes the event to listeners and listener, which subscribe to the event and get notified when something changes. In my prototype the kernel subscribes entities to the collision manager as an ICollidable object, then the entity has a method call on collide which gets called when it collides with another ICollidable object. Through the onCollide method I can enact many different behaviours within my entities, for example when the player collides with a patient and presses the interact key, the players current tool’s behaviour will execute. The kernel can also unsubscribe entities from the collision manager, which will stop any behaviour that the entity requires the onCollide method to enact. The key problem I have with the collision in my game is that when the player collides with an IImmovable entity (an entity that stops the player from moving when it collides with it) the player is pushed away from the immoveable entity. This is to prevent the player from always colliding with the entity when they first intersect with it. However, this becomes problematic, because when the player wants to interact with the patient, they must hold down the movement key and the interact key at the same time. This feels extremely clunky and not the desirable outcome that I want to produce for my game, however I could not fix this bug in time.



This is the implementation of a the onCollide listener method that is taken from the game engine code. If the player collides with an entity, the entity is passed through the parameters of this method and in this example, if the entity that the player collides with is a an IImovable object, then the player will be moved back to the last position before the collision occurred.

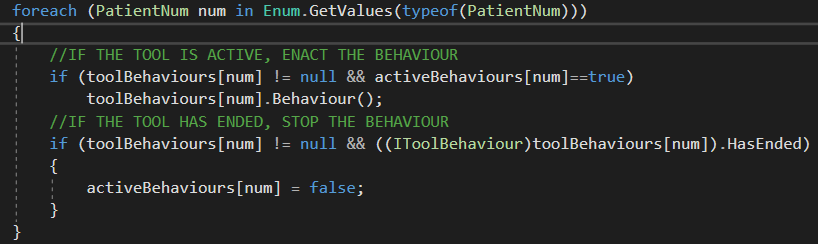
Behaviour Management

Class Diagram showing behaviour management of the tool behaviours



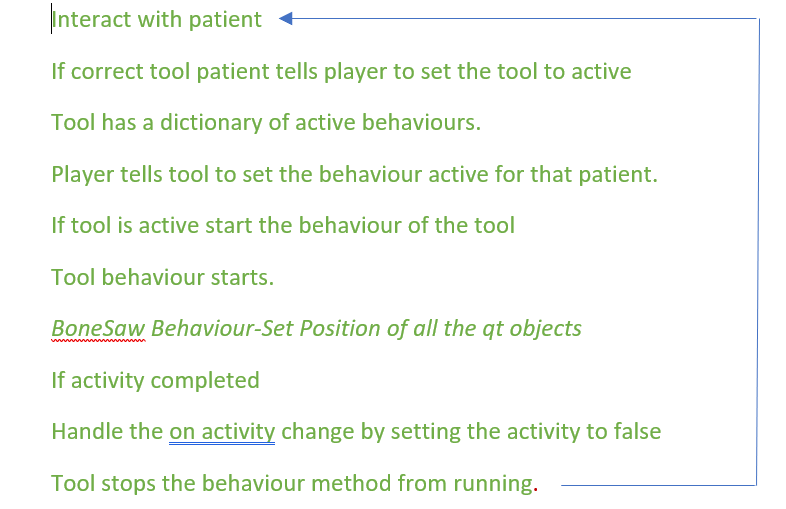
In my prototype, I have a behaviour for each tool. The challenge with this was switching between the different tool behaviours when the patient is cured/died. One of the functional requirements for the game was to have multiple activities occur simultaneously, which meant that I could not simply use one behaviour per tool(otherwise this would mean that I could not have 2 leech activities happen at the same time), instead I was forced to include a behaviour for each patient in each tool. The class diagram explains how I have accomplished this in my prototype.

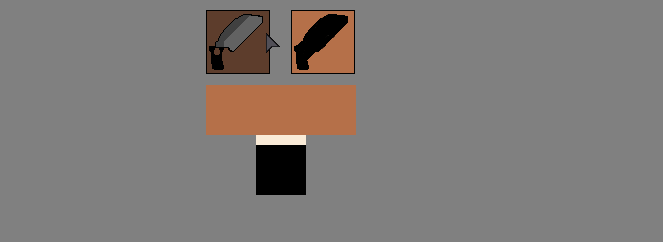


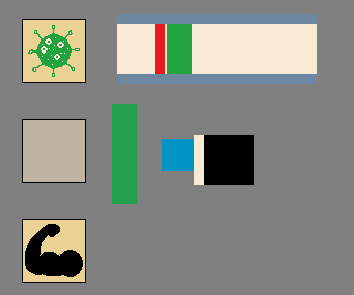


In the code above, I am storing all the active behaviours ( one for each patient) and if the behaviour is active, I am enacting the behaviour method in that tool behaviour. If the behaviour has flagged that it has ended i.e. the patient is cured/dead, then the behaviour method will stop running.

Structured English showing the program flow of my behaviour management with tool behaviours



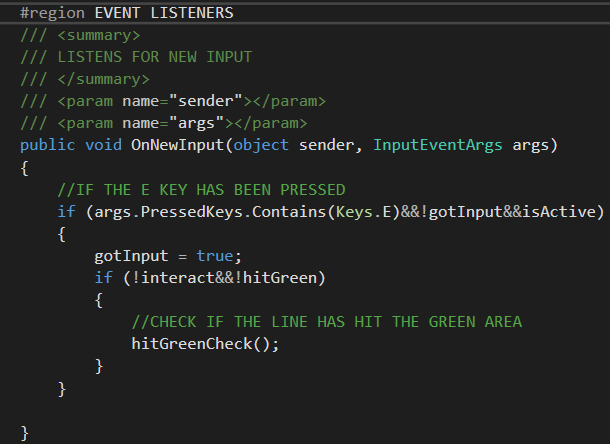
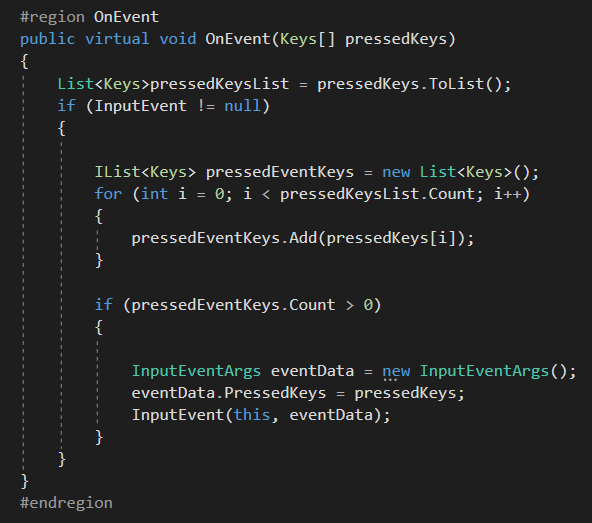
Input Management



The image on the right shows a patient (the blue square) and the player. I am interacting with the patient using the interact key, which is “F” in my game and then the activity (the quick-time event) is listening for the key “E” to be pressed, so that it can react in the appropriate way.

My prototype uses both keyboard and mouse input. The tool bench that the player is sat next to on the picture to the left has two buttons above it. These buttons listen for mouse input and add the tool to the player when clicked.

All the input management in my game goes through the architecture of my game engine. This uses event that the listener subscribes to. When a key is pressed, the input handler sends off an event to this subscriber. They then can choose what to do with this information.



The left image is the engine code for handling the keyboard input event and the right image is an example of where I have implemented this where the bone saw behaviour listens for the keyboard input that is fired from the engine code.

References

Carr, R., 2009. *Gang Of Four Design Patterns*. [online] Blackwasp.co.uk. Available at: <http://www.blackwasp.co.uk/gofpatterns.aspx> [Accessed 29th April 2020].