How to Develop for Cell City

Welcome to Team 42, the next big games studio! We are currently developing Cell City, a game designed to teach AP Cellular Biology in a fun way. Eager to start developing for this game? Great! Because this document is eager to teach you how!

Getting Started

The first thing you need to know is that Cell City is created using the Unity engine. We suggest familiarizing yourself with the basics of Unity first, but not too much since we will be explaining a lot of it in this document.

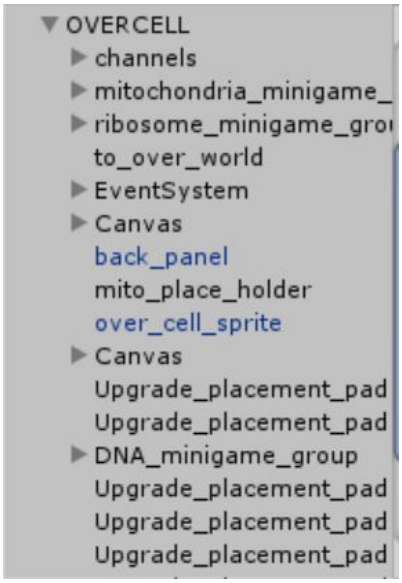
Alright, so let's get started. One of our major goals in Cell City was to make it so that all the organelle minigames (more on this in a bit) play simultaneously. In order to achieve this we had to make the whole game take place in one big scene, called MAIN_WORLD. So be sure to work on this scene when adding to the game.

Now, how does Cell City work? First I would suggest referencing the User Manual, or referencing the in game tutorial to get a good idea of how the game works. You manage a cell that has to collect and manage resources. These resources can then be used to upgrade your cell with different things, such as weapons. Think of the games Civilization and Simcity, but Cell City is much more fast paced.

As you probably know from Biology, a cell is made up of organelles that make the cell function properly. It is these very organelles that you use to help you manage resources. There is also the Over World aspect of the game that you use to gather Glucose, defeat enemy cells, and avoid dangers such as salt and water. So let's start from the big picture, the overworld and work our way down. But first, let's discuss some of the basic things we did to make development easier.

Container Objects:

Throughout the game we created many container objects to help make the hierarchy view in Unity much more organized. Think of them as folders. An example of this is:



This also has the added benefit of keeping objects in the same location relative to one another when the player cell, or other object, moves.

Tags:

You will find tags attached to many objects in cell city. For example, all Glucose objects are tagged with Glucose. This makes it much easier to detect objects in the game, or to check what two objects are colliding. As a result of us using tags you will often see code that gets, or checks an object's tag. We will go into more uses of tags later in this documentation.

game_master Object:

This object is the one of the main components of Cell City. The main function for this object is to store different values that are required to make the game function. A few of these values are Glucose, Water, Salt, etc. You can have a closer look at these values in the global_variable_test_script.cs script, which is attached to the game_master object. Not only does the game_master save values, it is also able to get values for other objects, which we will see different examples of later.

Displaying Amount of Resources:

At the top of the game display there is always a Canvas that shows the player how many resources they have. This Canvas has many child text boxes. These display

different values, such as the amount of Health, Glucose, or ATP a player has. These values are obtained from the game_master object, which we discussed earlier.

Player Movement:

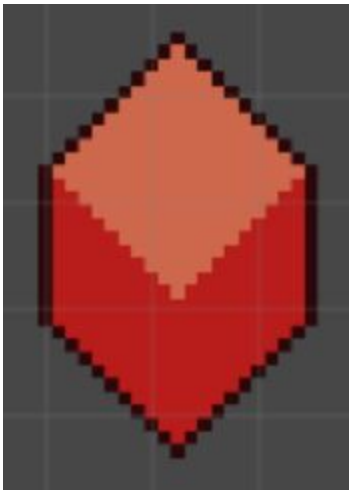
When in the overworld the player can move their cell using either WASD or the arrow keys. If you would like to work on the cellular movement you may reference the player_movement.cs file. This file is attached to the OVERCELL container object. In the Update function of the script, we check if the player is pressing down a key. If the player is we move the player cell in the appropriate direction. However, the player cell isn't the only thing that can move in Cell City, which brings us to the enemy cell.

Enemy Cell:

The enemy cell is a simple object. The game object is called Enemy Cell. All the enemy cell really does is look for Glucose in the overworld, and move towards it. Once the enemy cell touches a Glucose object (which will be discussed soon) it is destroyed. This is done using the attached Rigidbody 2D and Box Collider 2D on both the Glucose object and the enemy cell object. If you would like to take a look at how the enemy cell moves or finds Glucose, you can take a look at the evil_cell.cs file, attached to every Enemy Cell. The Glucose finding is under the FindGlucose() function, and the movement towards the Glucose is under the Update() function.

Another important part of the enemy cell is collision. As mentioned previously, the enemy cell collides with, and destroys Glucose. But the enemy must also be able to collide with the player's weapons (more on this later). Once the enemy cell collides with the player weapon the OnTriggerStay2D() function is called, and the enemy's Health value starts to go down. However, the subtraction of health is on a cool down, so that the enemy isn't killed immediately. This is done using the stopper function, which stops the enemy cell from getting damaged each frame, and is instead only damaged after a cool down period. Once the enemy cell's health value equals zero, the enemy runs the killSelf() function. In this function the enemy cell is destroyed and it spawns in a few Glucose objects. The Glucose then move to a random nearby location using a method in the Glucose object, that will be discussed very soon.

Glucose Object:



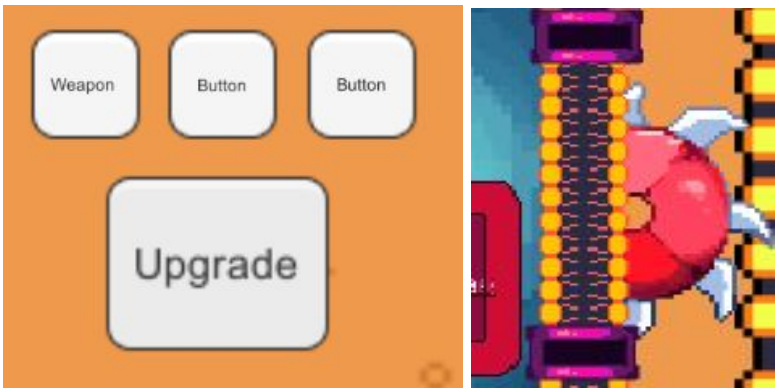
The Glucose object is very simple. It is only an object with a Box Collider 2D and Rigidbody 2D attached. These are used by the player cell object and enemy cell object when colliding with the Glucose. To aid in this collision we gave the Glucose object a Glucose tag. Whenever an object is collided with, we check if the object has a Glucose tag. If it does, then Glucose object is destroyed (and for the player cell, 1 glucose is added to the glucose value).

There is one other important part of the Glucose that we mentioned earlier, the random movement. This is done (as mentioned previously) when an enemy cell is destroyed. On destruction, the enemy cell calls the moveTo() function in the glucose.cs script (which is attached to every Glucose object). This function decides on a random distance to move, and a random direction. Then in the Update() function, the Glucose moves towards this location, one small step in every frame until it gets to the location. Once it reaches the target location, the Glucose object stops moving.

Salt and Water:

Throughout the Overworld you will notice two different objects, salt and water. These little patches of salt and water increase or decrease the amount of water in a cell (which is accurate to actual cellular biology). This was done by giving these objects Rigidbody 2D and Box Collider 2D components. So when the over_cell_sprite (the visual and collision part of the player cell) touches these objects the OnTriggerStay2D() function is called in the player_cell_collision.cs script, which is attached to the over_cell_sprite object. Depending on if the player is colliding with either water or salt the appropriate value is changed. However, there is a stopper value (which we went into more detail about under the enemy cell object) so that the values are not changed every frame.

Upgrades:



On the left is the user interface to create upgrades and on the write is the in-game instatited upgrades

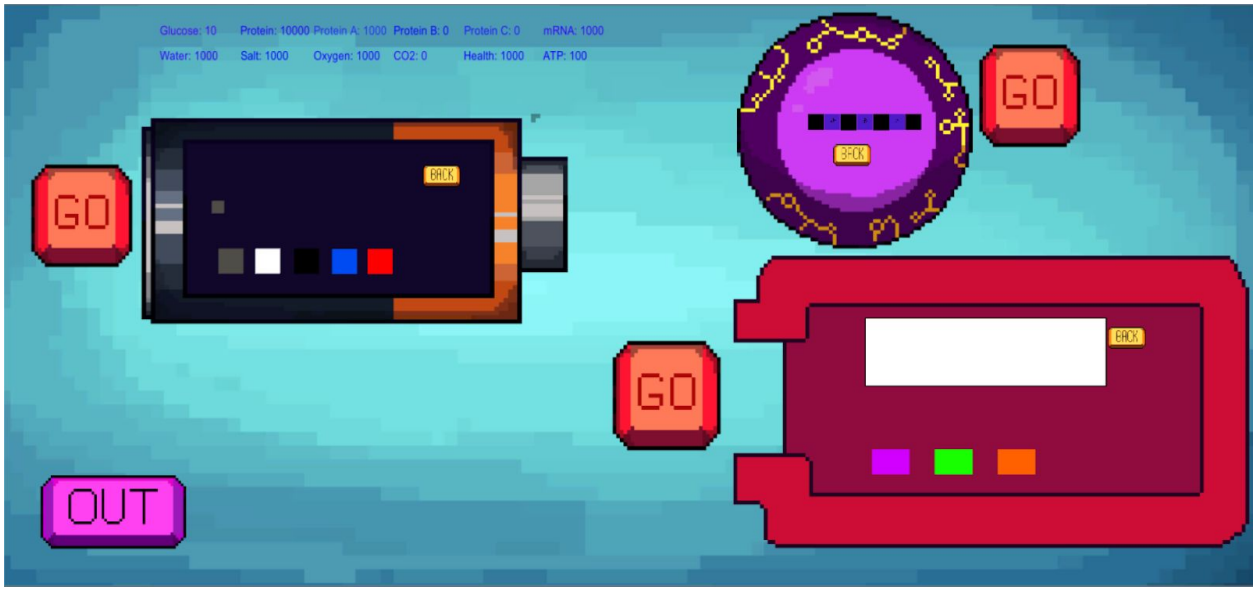
For the player to create upgrades they must interact with the overworld upgrade menu. The script that runs this component is the Upgrade Btn_1_Script.cs file. This menu is created through Unity's canvas utility and has its own template for developers to easily add on more possible objects and or upgrades for the player to instantiate. The basic layout is that each button creates a template object of the actual upgrade that the player wants to instantiate and it follows the player's mouse. This is done through ray tracing and calling the mouse's coordinates. For the template to turn into the actual upgrade object, the upgrade pad object must be clicked. This restriction is done by checking whether or not the mouse's coordinates are over the upgrade pad's coordinates. To further add more buttons and different upgrades, simply duplicate the existing buttons and change the object that it instantiates in the provided drag and drop development user interface. Each designated template object must also be duplicated from the base template object. To make the new upgrade object distinct, simply change the template and the new upgrade's render images to match what is desired and change the new template's transformation prefab to the actual upgrade object you want the player to instantiate.

Weapon Upgrades:

The weapon upgrade is represented by the weapon object and it allows the player to destroy enemy objects. The script that runs the weapon functionality can be found in the weapon.cs file and in the evil_cell.cs file. The weapon works by checking if there is an consistent collision with the enemy cell object and then runs its animation function while a collision exists. The actual damage to the enemy object is done in the evil_cell.cs script. The script checks if the enemy object is currently colliding with the weapon and

then reduces its health variable. An enumeration timer is then set off to spread the time between damage. This is done to avoid the amount of events of damage in a second equal to the framerate. Damage is further illustrated by having the enemy cell switch to a red sprite when it is damaged and then change to its normal sprite while the timer is ticking.

The Inner Cell



The inner cell area is where the minigames are housed, and where the majority of the resource value changing occurs. With this you can zoom in and out of minigames, play the minigames (each of which has different mechanics), and view tutorials the first time you zoom into a minigame.

Zoom:

```
// Update is called once per frame
void Update () {
    if (back_panel_script.turner==1 && stopper ==0)
    {
        transform.Translate(29, -8, 0);
        camera_component.orthographicSize = 7;
        return_full_screen.returner = 0;
        minigame = "ribosome";
        stopper = 1;
    }
    if (back_panel_script.turner == 2 && stopper == 0)
    {
        transform.Translate(-23,6, 0);
        camera_component.orthographicSize = 5;
        return_full_screen.returner = 0;
        minigame = "mitochondria";
        stopper = 1;
    }
    if (back_panel_script.turner == 3 && stopper == 0)
    {
        transform.Translate(18, 12, 0);
        camera_component.orthographicSize = 5;
        return_full_screen.returner = 0;
        minigame = "DNA";
        stopper = 1;
    }
```

```
if (return_full_screen.returner == 1)
{
    if (minigame == "ribosome")
    {
        transform.Translate(-29, 8, 0);
        camera_component.orthographicSize = original_orthograpic_size;
        back_panel_script.turner = 0;
    }
    if (minigame == "mitochondria")
    {
        transform.Translate(23, -6, 0);
        camera_component.orthographicSize = original_orthograpic_size;
        back_panel_script.turner = 0;
    }
    if (minigame == "over_world")
    {
        camera_component.orthographicSize = original_orthograpic_size;
        return_over_world.returner_over_world = 0;
    }
    if (minigame == "DNA")
    {
        transform.Translate(-18, -12, 0);
        camera_component.orthographicSize = original_orthograpic_size;
        back_panel_script.turner = 0;
    }
```

On the left is the “zoom in” state machine and on the right is the “zoom out” state machine

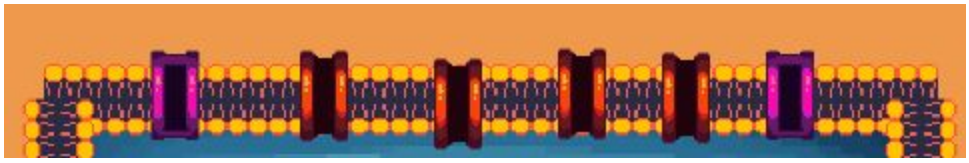
To have the player properly see and interact with the components of the cell, each minigame needs to have a zoom mechanic implemented. The script that dictates all camera movement is named camera_script.cs file and is a component of the camera object. The script has a basic state machine to dictate how the camera moves and zooms. There are two subsets of state machine in the main state machine controlling

the zoom. There are the “zoom in” states and the “zoom out” states. All zoom in states are triggered by the “GO”, and “ZOOM” buttons while zoom out states can only be triggered by the “BACK” and “OUT” buttons. The trigger scripts are in the back_panel_script.cs file and the return.cs file which can be found in the aforementioned buttons. For each zoom in state, there must be a subsequent zoom out state implemented. It is very important that only the relative coordinates are changed in order to ensure proper zooming after the player moves. The “zoom in” states are dependent on whether or not the player is currently in the inner cell perspective or the overworld perspective. Depending on which perspective, the turner variable in the triggers will either create no effect or cause the perspective to zoom into the dedicated minigame. The zoom out state machines works the exact opposite way, checking whether they player is in a minigame, which minigame it is, and then always zooming back into the inner cell or overworld perspective. Although triggers such as “GO” and “BACK” are instantiated from the same prefab, they create distinct trigger effects by having a tag that correlates to the minigame associated with each button object. The camera script checks this tag, checks its current state, and changes accordingly.

Tutorials:

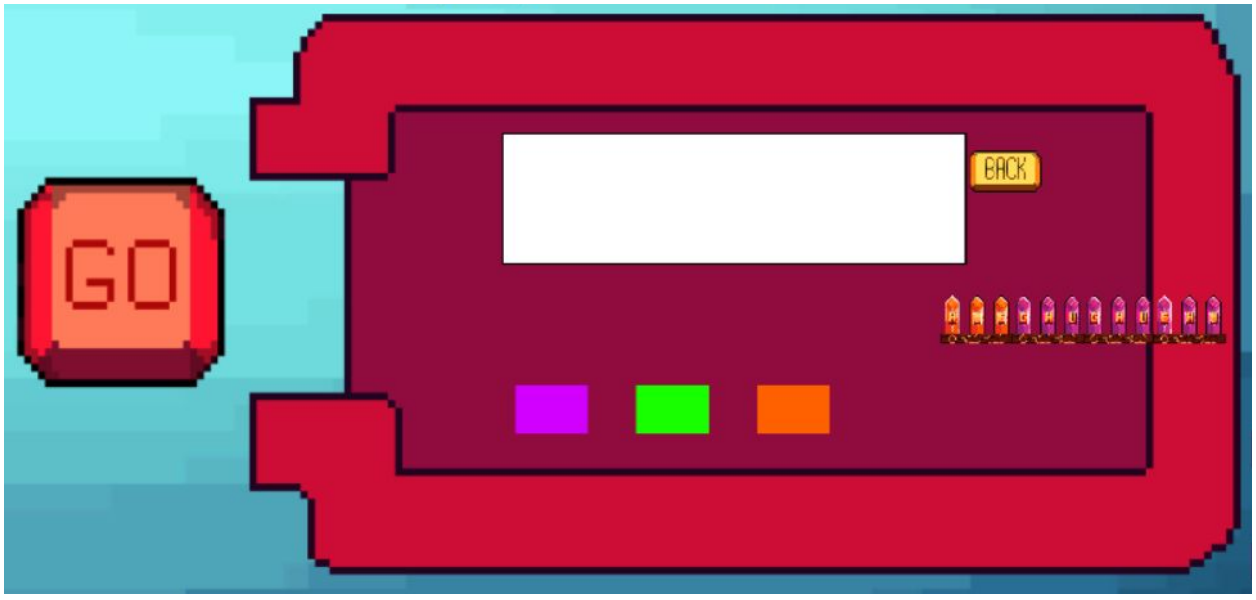
The first time the player zooms into a minigame, a tutorial appears, pausing the game until the user confirms they have read it by pressing a “Got It” button. The tutorial is a canvas with a textbox and button within it that is inactive when the game starts. The first time the player presses a zoom button for a particular minigame, pausing the game and activating the canvas. When the user presses the “Got It” button, the canvas is set to inactive and the game is unpaused. The tutorial only occurs the first time the player zooms in because of a boolean variable being set from false to true the first time it is clicked.

Lipid Bilayer Minigame:



The lipid bilayer component of the game involves the channel object, the game master object, and the over world cell object. The lipid channels have two states, open and closed. They change between these states by being clicked on. This process is dictated by a state machine. The channel also interacts with the game master object. The object mimics the process of being open or closed by manipulating the game master's "glucose intake" variable. As the the channel changes its state, it will either add or subtract one from this global variable, thus changing the amount of glucose the player cell collects when colliding with overworld glucose. The amount of glucose is determined by the glucose intake level multiplied by three and increased by one. As mentioned in a previous section, the lipid channels are created through player interaction with the upgrade interface.

Ribosome Minigame:

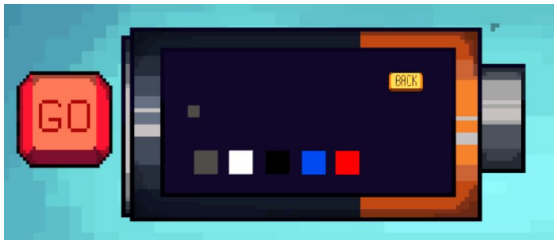


The ribosome minigame is a tower defense-style game where mRNA objects are traveling across the screen, and the player must click buttons to place cannons in a specific area to shoot them. The buttons, when clicked, set a variable to the type of cannon they produce, and when the player clicks on the area we specified in the code, the cannon is placed and the variable is set back to null. The mRNA are spawned from

an invisible Spawner object, which has an algorithm that chooses a “random” order for the final four mRNA in a strand (the first is always orange). They move across the screen, destroy themselves when they reach the edge of the minigame scene, and transform when they are hit with a shot from a cannon.

The main purpose of the cannon is to shoot “bullets” at the mRNA targets. The cannons are constantly look for mRNAs of their color. Once they are found, they are labeled as their target. The cannon then faces the target, and spawns in a bullet. The bullet then starts to move towards the target mRNA. Once the target is hit, the bullet is destroyed, and the mRNA’s tag is changed to inactive, adds a value to the player’s variable, and changes its sprite.

Mitochondria Minigame:



The mitochondria minigame is about placing objects in a certain order. Each object has a button which when clicked spawns a template for that object which follows the mouse position, rounded so that objects will stay in a grid pattern. The template, when the mouse is clicked on a valid area, places the object the template represents, as well as an InvalidSpace object, then destroys the template. The InvalidSpace is an invisible object with a BoxCollider2D, which will return a hit for the ray that is cast when trying to place an object, preventing another object from being placed on top of it. Each pad object has a 3D BoxCollider, which every frame checks on all sides for RaycastHits, and if a specific type of object is hit, the pad will transform, the final one producing ATP when transformed.

DNA Minigame:

This is the nucleus. This minigame is made up of mainly four black blocks that are clickable. Each of these blocks use the click_block.cs scripts. In these scripts we check for when the block is clicked. Each of the four blocks has its own unique tag, and depending on this tag we do different things. First the block is “activated”. Then the DNA Strand (with the strand_manager.cs script attached) container decides what to do with the clicked boxes. If the boxes surrounding A are clicked, then A is “activated”. If the boxes surrounding A, B, and C are clicked, then all of the letter blocks are “activated”. It is also here that we manipulate the player’s values held in the game_master object.

Miscellaneous

Pause:

When the “P” key is pressed, the Pause() function in the game_master object is called. When called, it checks what the current timescale is for the game. If it is not zero, the timescale is set to zero, preventing any automatic movement or automatic value changes in the game_master. We have placed code in the Overcell object which prevents player movement while the game is paused as well. If the timescale is zero and the pause function is called, the timescale is set back to its default value of one, unpausing the game.