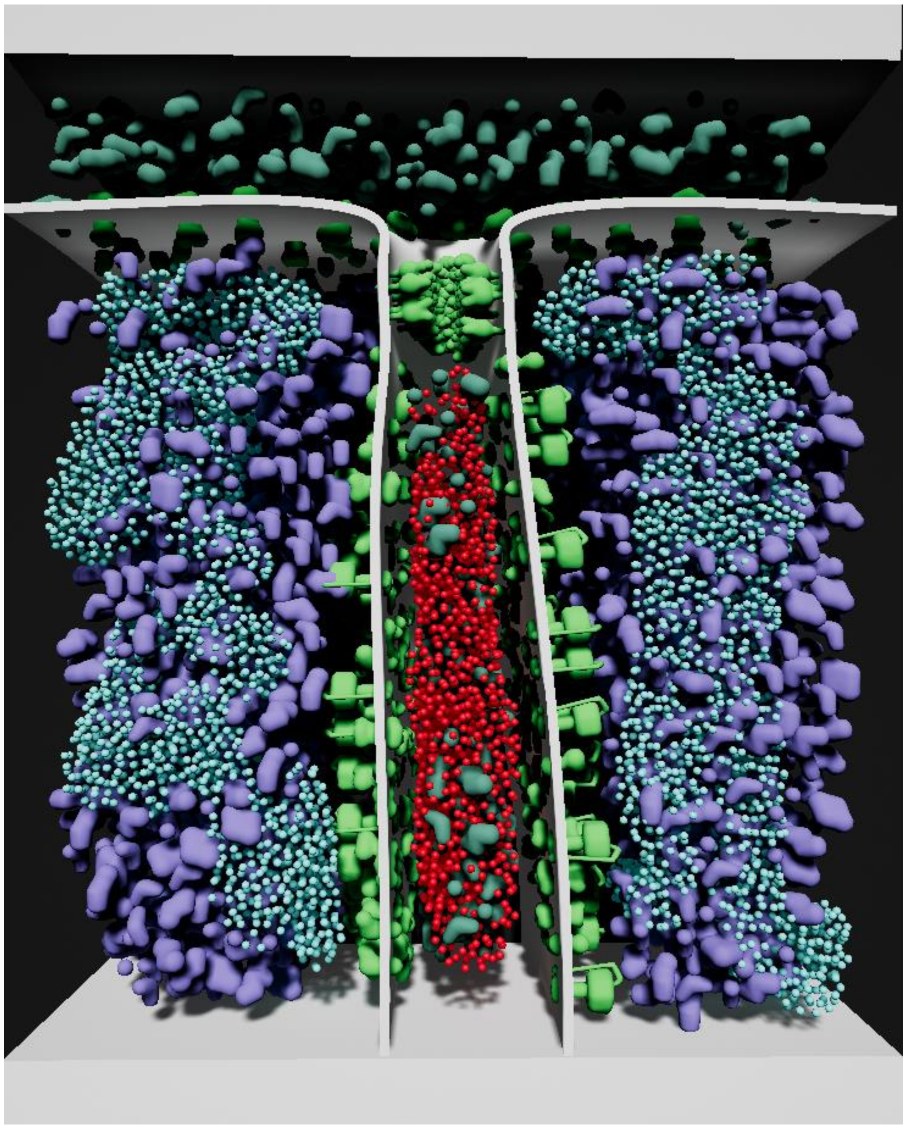
# Lifebrush API Documentation



Lifebrush API Created By: Dr. Timothy Davison
Documentation By: Michael Wahba

## Contents

## 1 Introduction

Features I added that I need to document

1. Guided Lectures
2. Info Pane
    a. put info pane into scene
    b. Set a title
    c. select which actors you want to include
    d. add information tp actors
3. Snapshot Slideshow

## 2 Terminology

Lifebrush is a complex, interactive multi-agent system which has many moving parts. To minimize confusion, it is a good idea to familiarize yourself with the terms we will be using in this document. The best analogy is to think of Lifebrush as a painting simulation. Like a painter we have paints, a palette and a canvas to paint on.

1. **Agents**
   a. Agents are like our paints. Using the controller we select which agent we wish to paint with and populate our canvas with it.
2. **Palette**
   a. In Lifebrush, the palette is analagous to an artist's palette, it is where we store our painting materials. However, in Lifebrush rather than using paints we are using agents, each with their own behaviour.
3. **Canvas**
   a. Like a painter, the canvas is where we paint. In Lifebrush, the canvas is the region of where we can place agents.

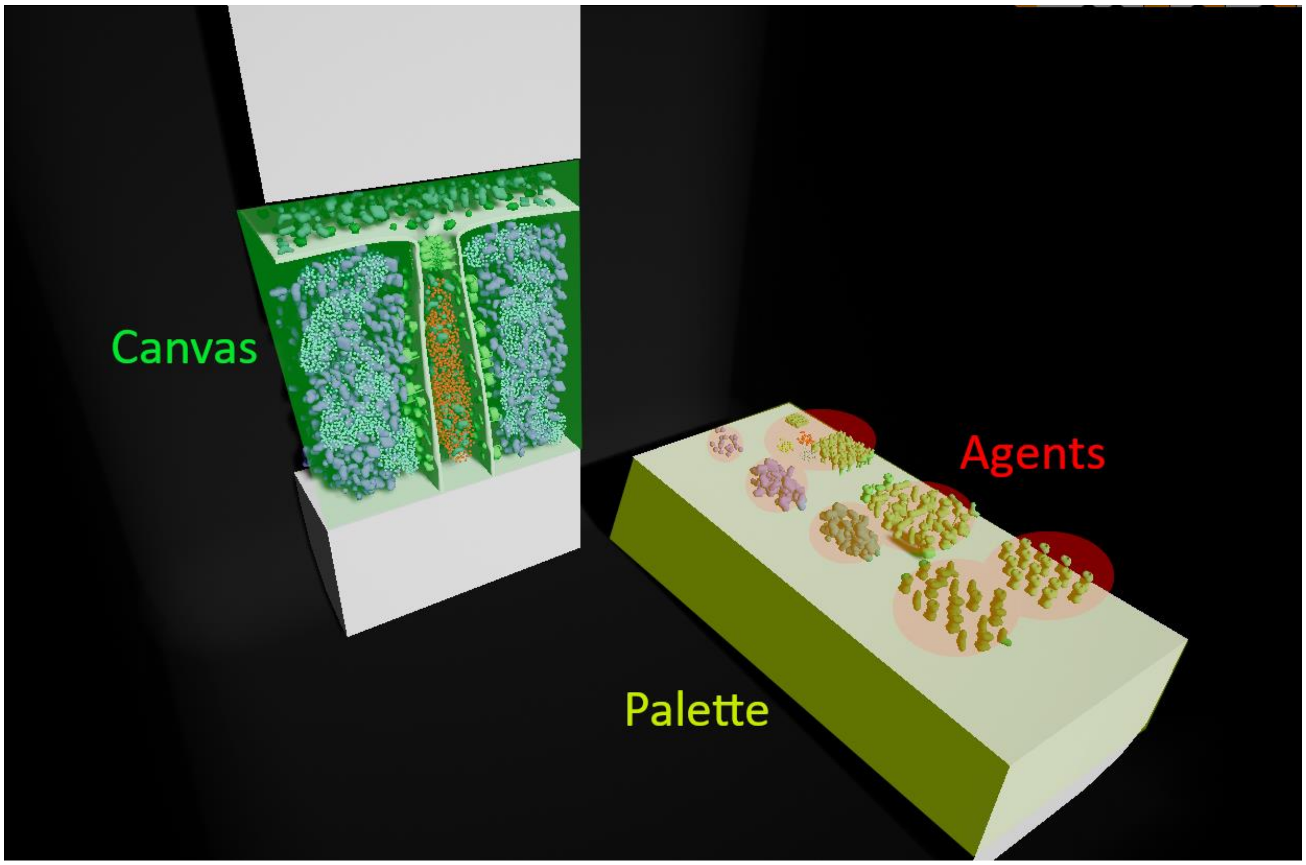


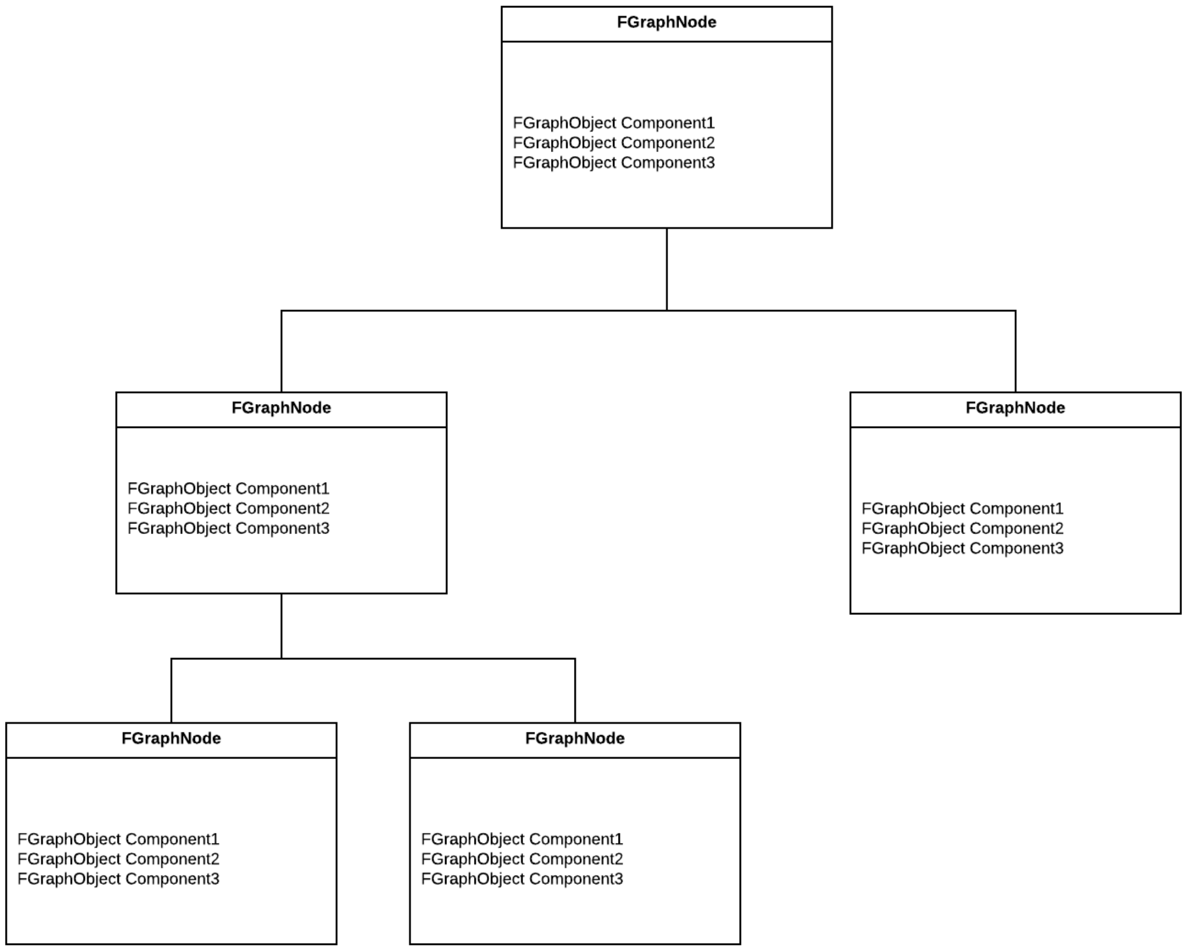
*Figure 1 Overview of Lifebrush work area. Canvas in green, Palette in yellow, Agents in Red.*

## 3 Graph-Entity-Component System

Lifebrush utilizes a Graph-Entity-Component System (gECS), a modification on a standard Entity-Component-System (ECS) typically used in game engines like Unity or Unreal. The standard Entity-Component-System is great for most use cases but is not feasible when we are using tens of thousands of entities, as we are with Lifebrush. For Lifebrush to be an effective tool it must efficiently render the relationships between thousands of molecules.

| **Entity-Component-System** | **Graph-Entity-Component-System Equivalent** |
|---|---|
| Entity | Graph Node |
| Component | Graph Object |
| | |

The solution is to add a layer of topology to the entities in a scene.



You can think of each Entity or Agent in your simulation as a node on a graph. Each **Graph Node** can be a single molecule or a cluster of molecules that behave to its own specific ruleset. Inside each Graph Node are **Graph Objects.** Graph Objects are like the components in a standard ECS and are added to each Graph Node to add certain behaviours. For example, the Velocity Graph Object gives a Graph Node a linear or angluar velocity in 3D space.

Lifebrush operates on a real-time simulation graph data structure to simulate large-scale agent-based models. In the gECS, each agent is a graph vertex and the relationships between agents are edges.

The implication of this system is that when we are writing functions which add or remove agents from the simulation, we need to interface through the graph itself. Luckily, any interaction with the graph is abstracted away through the Lifebrush API, however there are some key classes that we need to understand to use Lifebrush effectively.

## 3.1 Key Lifebrush Classes

Outlined below is just a few classes in the Lifebrush API that you will likely use. Note that underneath each class is only a **few** properties and methods that will most likely come helpful. As mentioned above, many operations involving the actual topology of the gECS has been abstracted away, thus to you, the user, you will only have to be familiar with the interface classes and methods that will help you achieve your goal.

**A Note on Unreal Engine's Naming Conventions:** Unreal Engine automatically prefixes class names with a letter to distinguish them from variable names.

| Prefix | Type |
|---|---|
| T | Template Class |
| U | Classes that inherit from UObject |
| A | Classes that inherit from AActor |
| E | Enumerations |
| S | Abstract Interfaces |
| F | Type Definition of a Struct |

## 3.2 Graph Classes

The classes outlined below make up the graph structure of the Lifebrush simulation. Typically, you will not need to worry too much about the topology of the graph, as it is abstracted away. Some of these classes are used as base classes for the objects that will comprise the simulation. You will likely not be using these classes as base classes in your custom classes.

### 3.2.1 FGraph

Inherits: NONE

Properties

- allNodes:TArray<FGraphNode>

Methods

- beginTransaction(): void
    - Call this method before you make alterations (adding/removing nodes) to the graph

- endTransaction(): void
    - Call this method when you are done altering the graph
- componentStorage(ComponentType): FComponentStorage&
    - This method lets us access the nodes, in the graph, corresponding to a specific component type.

Description

The FGraph class outlines the graph data structure that will contain all the agents as nodes. There are many methods that work behind the scenes to add and remove nodes to the graph, however as a user you will not have to worry about these topological considerations.

### 3.2.2 FGraphNode

Inherits: NONE

Properties

- Id: int32
    - A node's unique id
- position: FVector
- orientation: FQuat
- edges: TArray<int32>

Methods

- hasComponent(ComponentType): bool
- addComponent(FGraph&, ComponentType): FGraphObject*
- removeComponent(FGraph&, ComponentType): void
- isValid(): bool

Description

The FGraphNode describes the actual nodes that will populate our FGraph. The FGraphNodes are analagous to the entities in an entity-component-system. The components we add to the nodes are custom classes that inherit the FGraphObject class.

### 3.2.3 UGraphSimulationManager

Inherits: UObject

Properties

- simulations: TArray<UObjectSimulation>

Methods

Description

The simulation manager keeps track of all the attached simulations that Lifebrush is managing. The simulations attached will be of the UobjectSimulation class (see below) and each simulation runs independently of each other. The Graph Simulation Manager is in charge of "ticking" each simulation in turn, which then ticks each graph object that a corresponding simulation needs to tick. (i.e. The Central Dogma simulation will only tick the graph objects that are referenced within that simulation.)

## 3.3 Interface Classes

The classes below are the ones you will likely use most often to create new Lifebrush simulations. The following classes will form the base classes for your agents and the simulations which they belong to.

### 3.3.1 AElementActor

Inherits: AStaticMeshActor

Properties

- graphObjects: TArray<FTimStructBox>

Methods

Description

The Element Actor is the class which acts as the Graph Nodes in our simulations. The Element Actor objects will make up the palette, which you will use to paint your simulations. Each Element Actor class has a set of Graph Objects, which confer behaviours to that actor (i.e. The Random Walk Graph Object will cause the Element Actor to move randomly in 3D space). Most likely, you will add Element Actors and their corresponding Graph Objects to your scene via the Unreal Editor. However, we can also instantiate new Element Actors in our scene via the Lifebrush API.

### 3.3.2 FGraphObject

Inherits: NONE

Properties

- nodeIndex: int32

Methods

- nodeHandle(): FGraphNodeHandle
- isValid(): bool

Description

We inherit from the FGraphObject class when we want to create unique components for the agents in our simulation. The node index is a unique identifier for a given agent in the simulation. When programming behaviours for our graph objects it is important to make sure the object is valid first. We do this using the isValid() function.

### 3.3.3 UObjectSimulation

Inherits: UObject, ComponentListener

Properties

- graph: FGraph
- simulationManager: UGraphSimulationManager

Methods

Description

The UObjectSimulation is the class we inherit from when we want to create our simulations. One way to think of it is that the UObjectSimulation is the container which holds the graph objects.

# 4 Lifebrush Workflow

## 4.1 Overview

Lifebrush operates by using user-defined simulations to control the agents onscreen corresponding to a give simulation. Using the Lifebrush API we can define the parameters of our simulation and specifically single out which agents we want our simulation to act on. Once the simulation is set up, we then need to populate the in-game agent exemplar palette with agents we can use to paint. From a high-level using Lifebrush can be reduced to three steps.

1. Setting Up A Scene
2. Creating Custom Simulations and Agent Behaviours
3. Populating the Exemplar Palette

Once these three steps are taken care of, then you are all good to go!

## 4.2 Setting Up A Scene

There are a few things that need to be present in your Lifebrush scene for the simulation to work:

- A Simulation Actor which holds the flex simulation component
- The VRSketch Pawn to enable user interaction
- The Exemplar Actor where we store our agents

Luckily, included in Lifebrush is a “Template_Level” default level which contains these necessary components and nothing else. Starting with the Blank Template_Level is the ideal way to start building your own Lifebrush simulations. You can find the Template_Level in the “Scenarios” folder in the Unreal Editor content browser.

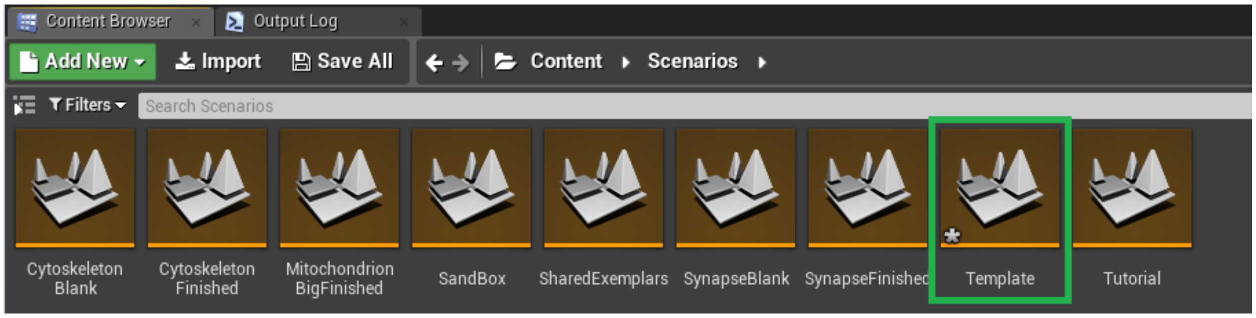
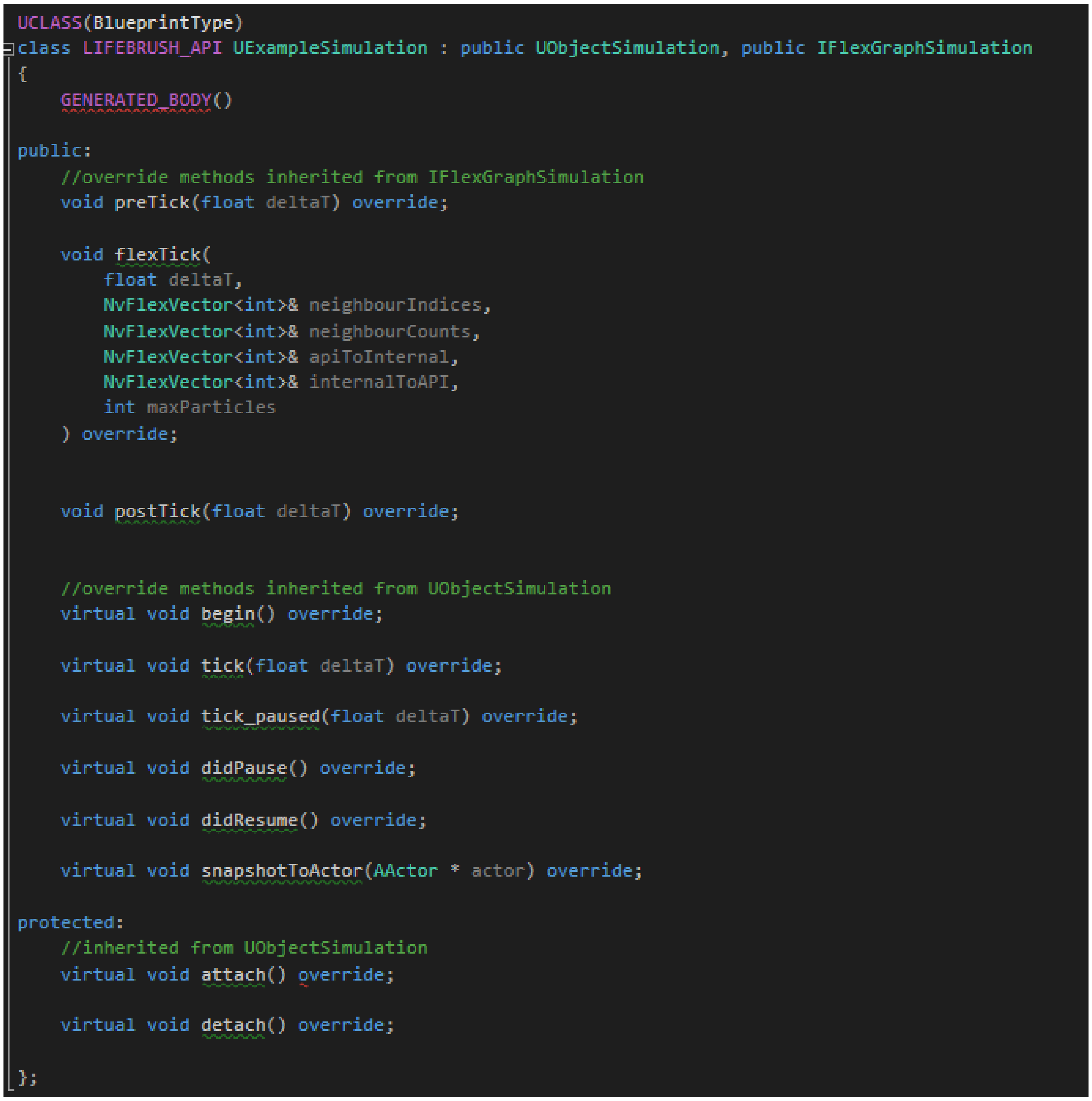


*Figure 2 The Pre-made Template level will start you off with everything you need to easily start building simulations for Lifebrush.*

## 4.3 Creating Custom Simulations and Agent Classes

To create your own simulation in Lifebrush, you must create a class that inherits from the “UObjectSimulation” and “IFlexGraphSimulation” classes. This class will be the object that “ticks” each individual object associated with it. Figure 3 shows an example of how to implement a simulation class as well as the inherited methods at your disposal. The most important inherited method is the FlexTick() method which ticks all the Flex particle objects associated with a given simulation. We will go into more detail on FlexTick() later.

Our Simulation can be thought of as the brain which coordinates the actions of all our agents. For example, if we want to make a simulation of the activity of the mitochondria, the simulation will keep track of each ATP molecule and the state of each ATP synthase and trigger the conversion of ATP to ADP when certain conditions are met.



```
UCLASS(BlueprintType)
class LIFEBRUSH_API UExampleSimulation : public UObjectSimulation, public IFlexGraphSimulation
{
    GENERATED_BODY()

public:
    //override methods inherited from IFlexGraphSimulation
    void preTick(float deltaT) override;

    void flexTick(
        float deltaT,
        NvFlexVector<int>& neighbourIndices,
        NvFlexVector<int>& neighbourCounts,
        NvFlexVector<int>& apiToInternal,
        NvFlexVector<int>& internalToAPI,
        int maxParticles
    ) override;


    void postTick(float deltaT) override;


    //override methods inherited from UObjectSimulation
    virtual void begin() override;

    virtual void tick(float deltaT) override;

    virtual void tick_paused(float deltaT) override;

    virtual void didPause() override;

    virtual void didResume() override;

    virtual void snapshotToActor(AActor * actor) override;

protected:
    //inherited from UObjectSimulation
    virtual void attach() override;

    virtual void detach() override;

};
```

*Figure 3 Example Lifebrush Simulation class. Also included are the overridden methods.*

After setting up the simulation, we now need to set up our simulation actors. The process is very similar to how we set up our simulation.

Our agent classes will inherit from the "FGraphObject" class. The FGraphObject class does not have any virtual methods that must be implemenented, but it does have some properties that may come in handy. Those properties can be viewed in section 4.1.2.

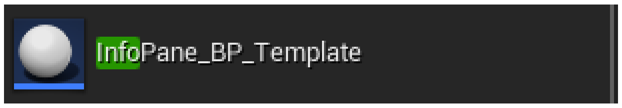
### 4.4 Guided Lectures

### 4.5 Info Panes

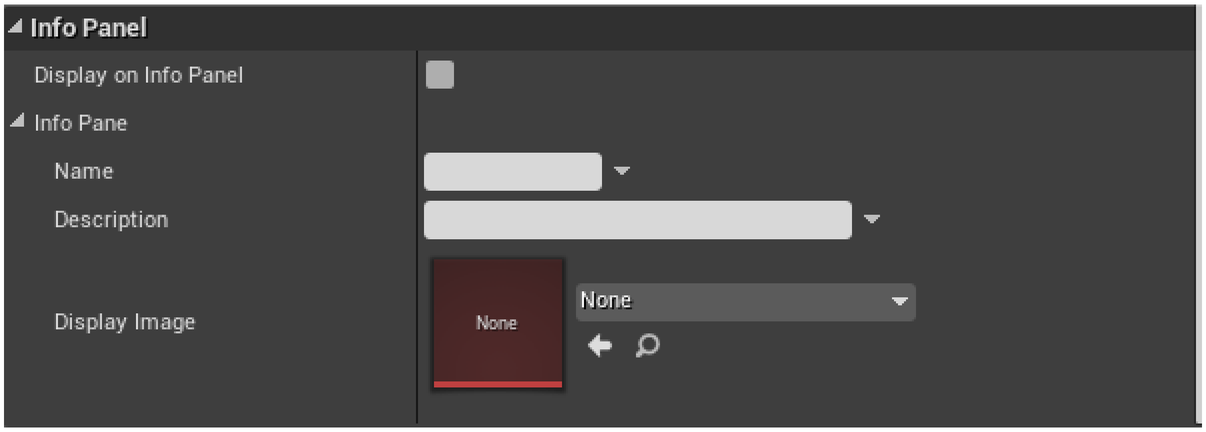
## 5 Lifebrush Features

### 5.1 Information Pane

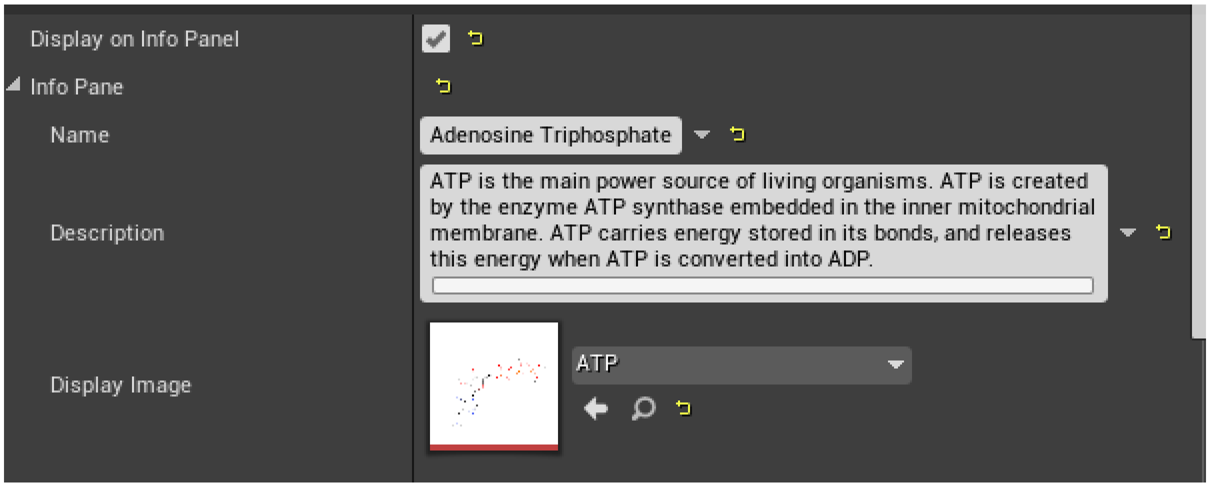
The Information Pane object is very useful for providing background information on the actors used in your palette. To place the Information Pane in your scene, all you need to do is drag and drop the following Blueprint into your scene:


InfoPane_BP_Template

Once the InfoPane object is in the scene, it will automatically find all the Element Actors in the scene so you do not need to worry about configuring that. All you need to do know is fill out the "Info Panel" section on each Element Actor you wish to be compatible with the Info Pane.



The "Display on Info Panel" parameter is a boolean which dictates whether or not that Actor's information will show up on the Info Panel. The "Name", "Description", and "Display Image" fields are used to fill in the information you wish to show up on the pane.



Display on Info Panel ✓

Info Pane

Name: Adenosine Triphosphate

Description: ATP is the main power source of living organisms. ATP is created by the enzyme ATP synthase embedded in the inner mitochondrial membrane. ATP carries energy stored in its bonds, and releases this energy when ATP is converted into ADP.

Display Image: ATP

Once these fields are filled in, when the user selects this agent in the exemplar then the filled in fields will automatically show up on the information pane in-game. If the user selects an agent which does not have “Display on Info Panel” checked, then the panel will disappear until another agent is selected.

## 6 References and Further Reading