Point And Click Engine

Documentation

Introduction

P.A.C. Engine is a product aimed at developers, and focused on providing all the necessary tools to expand upon when developing a point and click game. Please pay attention to the fact that this asset is merely meant to serve as a basis for your game, as games in this genre vary widely in terms of features and overall mechanics. New features will be added incrementally in the future to provide a wider array of possibilities when it comes to creating your own games with P.A.C. Engine.

Before we begin

Backups: Please make sure to always backup your projects. Although this may seem obvious, frequent backups can save you an incredible amount of time in case of data loss at any point in the project. Additionally, do not hesitate to acquire an external hard drive to use for your backups, as this way save you a lot of trouble in case of machine hardware failure. Cloud backups are not recommended, especially for large commercial projects due to risks of code / data leakage.

Additional features: P.A.C. Engine is designed to provide the majority of tools, essential in an development of a point and click game. However, please take into consideration the fact that most p.a.c. titles differ in terms of functionality etc., thus for any specific feature requests do not hesitate to contact me directly.

Errors: Since the asset is in early stages of development, you may encounter unexpected behaviour or errors. In such an events, please contact me immediately and the issue(s) will be resolved as soon as possible.

Demo

Before jumping into analyzing the inner-workings of the asset, I highly recommend setting up and testing the demo first.

First you will need to add all three of the demo scenes to the build settings. Please open the **Initiator** scene, then go to **File -> Build Settings** and click on **Add open scenes.** Once you are done, do the same for the **Demo1** and **Demo2** scenes.

Once you have added all three scenes to the build settings, please click on any object on any scene, then in the top right corner of the inspector you will find a dropdown of layers. Click on the dropdrown and then click on **add layer.** Afterwards, add two new layers into any of the empty fields. The first should be called **CollisionLayer** while the second one **MovementCatcher.** Now, open the **Initiator** scene, expand the **canvas** object and assign the **MovementCatcher** layer to the object with the same name. Now open both the **Demo1** and **Demo2** scenes, and assign the **CollisionLayer** to all object that the player is supposed to collide with. For this demo, assign the layer in question to every object under the **GroundColliders** object and to the **NPC** object under **Objects / Entities.**

In general, whenever you create an object that is supposed to collide with the player, make sure to assign the **CollisionLayer** layer to it.

Lastly, click on the **Canvas** scene on your **Initiator** layer and select **UI** as the sorting layer.

To launch the demo, simply open the "Initiator" scene (inside the scenes folder) and hit play. Take your time to look at all the various objects and scripts present at the scene.

Structure

Before jumping into details, it is important to understand how the asset is operating.
1) Starting level (initiation): The first scene of every game made with P.A.C. Engine must always contain several game objects which are designed to be persistent and are comprising the core of the engine. Namely:

a) Main Camera: A camera which is initiated once in the entire game and allows the viewing of any scene.
b) AudioHolder: Which is an object designed to contain AudioSource elements to play various audio throughout the game.
c) Event System: Which allows interaction with UI.
d) Game Manager: Which is the heart of the engine and hosts the majority of the scripts used by the asset.
e) Player: A player object which is initiated only once and persists throughout the game.
f) Canvas: Which hosts all the user-interface related objects.

In addition to the latter, a non-persistent Level Manager object, which is unique for each scene, should exist on the scene to initiate basic starting values.

2) Other levels: Since almost everything is initiated on the initial level, post-initial levels are meant to exclusively host in-level assets such as objects, entities (interactable objects or NPCs) and items. Please pay attention to the fact that post-initial levels should not contain camera, player or UI objects.

GameManager

Like previously mentioned, the game manager object is responsible for hosting all scripts vital to the operation of the asset. Since understanding the way in which the latter scripts operate is essential to understanding P.A.C. Engine, let's take a look at each of them one by one.

1) Persistence creator: This script is attached to all object that should exist between all levels without the need of re-initiation (i.e. these objects are made persistent).
2) Movement: This script is responsible for handle player movement. More specifically, it takes care of collision detection, actual movement and animation.
3) Player Manager: Is designed to allow easy management of core player information. It is heavily advised that this script is used to manage any future player data added for any of your projects.
4) Database: The database is designed to contain the majority of game related information. The database will be analyzed in-depth later on.
5) Support functions: In order to avoid "copying and pasting" functions across the asset, this script is designed to host any functions shared between more than one other script.

6) Speech: Since P.A.C. Engine used a “bubble” speech system (one in which text appears above the game object “talking”, this script is designed to handle the spawning and other actions related to the creation and “destruction” of said text.
7) Entity Interaction Manager: A simple script that allows or blocks entity interaction.
8) Reference Holder: In order to avoid creating multiple object references across various script, this script was created to be a collection of such references (allowing other scripts to easily access these objects).
9) Camera Position Monitor: This script manages camera position. By default, camera always follows the player. Boundaries can also be set in order to block the camera from going “out of frame”.
10) Inventory Generator: A script that is responsible for generating the inventory based on items contained in the database.
11) Quest Generator: Responsible for generating a list of all quests.
12) Journal Generator: Responsible for generating a list of all journal entries (things that entities have “said”).
13) Background blocker: A simple script that block any background activity (entity interaction, movement etc.) when UI is active.
14) Level Loader: A script designed to load levels based on level ids in the database.
15) Save/Load: A script hosting save & loading features/
16) Cursor Changer: A script hosting cursor related functions.

Database

The database allows easy access and modification of in-game data such as quests, items, music etc. By default the following data can be accessed and modified:

1) Items: An array of all in-game items. The customizable options for each item are the following:
   a) Name: The name of the item.
   b) Id: A unique item id.
   c) Icon: item icon in the inventory.
   d) Acquired: does the player have the item ? - If so, it will be displayed in the inventory.
   e) Use Function: A function from ItemActions.cs (by name) that will be triggered in the item is selected in the inventory and the “use” button is clicked. In other words, an action to be executed when the player uses the item.
   f) Quantity: How much of the item does the player have.
   g) Examine text: An array of lines of text to be displayed when the item in “inspected”. The latter text will appear in a separate box to describe the item.
   h) Combinations: A list of items that the item in question can be combined with an the combination results. For each “combination” you can indicate the ID of the item with which the current item will be combined, the product item ID (both ids should me item ids in the database), and whether the original or the product item should be destroyed after the combination.

2) Entities: A list of all in-game entities. An entity can have a unique ID, name and stage ID. Each stage ID should be unique since stages indicate story progression. See “stages” section.
3) Quests: A list of all in-game quest. All options are pretty self-explanatory.
4) Journal: A list of all journal entries. This list is being filled by default after an entity has made dialogue with the player.
5) Locations: Essentially a list of all levels. The main reason this list exists, is to create a flexibility when it comes to adding new levels or deleting existing ones. Each level has a name, a unique ID and a build settings index which can be changed at any time. When creating your game, I highly recommend accessing levels using their ids in the database and retrieving the build settings index via the appropriate function in “supportFunctions.cs”, instead of simply using the build settings index.
6) Music: A list of all background music. Each entry should have a unique id and an audioClip.
7) Ambience: A list of all ambient sounds, identical to the music list.

Items

Items are in-game objects that can be picked up, examined, used and combined. In the database section, we covered how items are registered in the database, but making an in-game object into an item is different. First and foremost, any on-scene “item” should be a an object outside of canvas (as it is not considered UI), with a box collider 2d component attached to it. Additionally, an item should have a sprite renderer and the item.cs script attached in order to make it visible and interactable. The item.cs script should be configured as follows:

a) Id: Is the unique item id in the database.
b) Instance Id: A unique id which is not present anywhere else. This id is used to mark item instances as “picked up” when necessary in order to destroy any items that have already been picked up when the scene is loaded for a consequent time.
c) Can be picked up: A variable allowing to indicate whether the item can be picked up. Although it may seem as a redundant feature, the latter allows more flexibility when writing custom scripts. For example, if an item should not be pickable unless the player interacts with an entity prior, initially the variable may be false and once the player does interact with the entity in question, it may turn to true.

Lastly, all on-scene / in-game items should be tagged as “Item”. Once the item has been configured, it should be pickable. Once the player has picked up the item, they should be able to view it in their inventory as well as, inspect, use, and combine the item based on how was set up in the database.

Entities

Entities are interactable in-game object that can player various roles game-wise. For instance, an entity can be an NPC, an animal or even a simple door.

To create an entity, create a non-canvas object with a 2d collider attached to it, tag the object as “Entity”, set the layer as “collisionLayer” if the player is supposed to collide with the object, and lastly add the Entity.cs script to the object.

The entity script is visually divided into three sections (tabs) to make navigation and editing easier.

Entity.cs Breakdown:

1) Entity ID: An entity ID matching an entity registered in the database.
2) Can Be Interacted With: Can the player interact with the entity ?
3) Stages: Stages are essential interaction starting points. Each stage has a unique id and a starting node value. The id of the stage is recorded in the database to keep track of story progression. The starting node id is the id of the node which is going to be accessed when the object is on the stage in question and the player interacts with the object.
4) Options: Various options can be displayed to the player to choose from during interaction. Each option will eventually lead to a different node (aka a different set of actions). However, please keep in mind that the number of options that can be spawned at once is limited by the amount of option spawn points under **Canvas->Options->OptionSpawnPoints.** You can add as many additional option spawn points as you like for your game. An options is broken down like this:
   a) Option id: Is a unique id.
   b) Option text: The text displayed in the options menu.
   c) Go To Node On Click: Should a node be displayed if the player clicks the option?
   d) Node ID: The id on a local node (on the same entity)
5) Nodes: Each node is a collection of possible actions. You may choose to give an item to the player via a node, display dialogue or even both. Nodes can be also used to check whether the player has met certain conditions before allowing the player to interact with them properly. Here is a quick breakdown:
   a) Node Id: A unique ID, also used by options when specifying nodes.
   b) Display text: If selected, text will be displayed in a speech bubble above the selected target.
   c) Text Target: A gameObject on the coordinates of which the text will be displayed. I highly recommend creating a separate, empty gameObject for each entity just for the sole purpose of being the text spawn point.
   d) Text: The Actual text to be shown.

e) Player Is Talking: Is the player the one doing the talking ?
f) Talking entity id: The ID of the entity talking. This field should only be specified if the player is **not** the one doing the talking.
g) Invoke Custom Events: Should any custom functions from EntityCustomEvents.cs be executed ? - Functions should be simply specified by name without parenthesis ("function()" should be specified as "function").
h) Events to be invoked: If the latter was set to true, you can specify the function names to be invoked in this array.
i) Play Audio with text: If you game has voice-over and you wish to play audio along with your text, simply set this value to true.
j) Audio: The Audio to be played.
k) Conditions to check: An extensive list of conditions to be checked before performing any other actions on the node. The condition checks are as follows:
    i. Check Quest Exists: Check whether player has acquired a specific quest.
    ii. Quest exists ID: The id of the latter quest.
    iii. Redirection node if exists fails: Node to be accessed if the latter check is not satisfied.
    iv. Check quest completed, id and redirection node: Identically as before, check if the player has completed a certain quest and if not, redirect to another node.
    v. Check item acquired, id and redirection node: Check if the player is in possession of a specific item, and, if not, redirect.
    vi. Check item quantity, id, required quantity and redirection node: Check if player has more than a specified quantity of an item before progressing.
l) Go To Node On Click: Access a specific node after accessing this node ?
m) Go to Node Id: The id of the node to go to.
n) Display options & display options ids: Whether options should be displayed when the node is accessed and a list of options to be displayed.
o) Set New Stage id: Should we set a new stage id in the database for this specific entity ?
p) New stage id: The id of the stage to be set.
q) Items to add: A list of items to add to the inventory and their quantities.
r) Item to remove: A list of items to remove. Remove all removes the item completely, while removing a quantity allows to remove a specific quantity of the item.
s) Quests to get & Quests to complete: An array of the quests to get (by ids) and an array of the quests to complete.
t) End Dialogue on Click: End dialogue after clicking or accessing the node.
u) Load level: Load another level ?
v) Load level id: The id of the level to load in the database.
w) Load Spawn X & Y: The player-spawn-coordinates on the newly loaded level.

XML Import / Export
XML import and export allow easy backup and manipulation of data. Currently, P.A.C. Engine allows the import and export of most "node" data. The import and export button, along with the path and name field are located at the bottom of each entity script.

Be very careful when importing node data since the imported file will always overwrite the nodes list completely.

To add data to your xml file, simply copy the last line of node data, paste, and edit it to create a new list element.

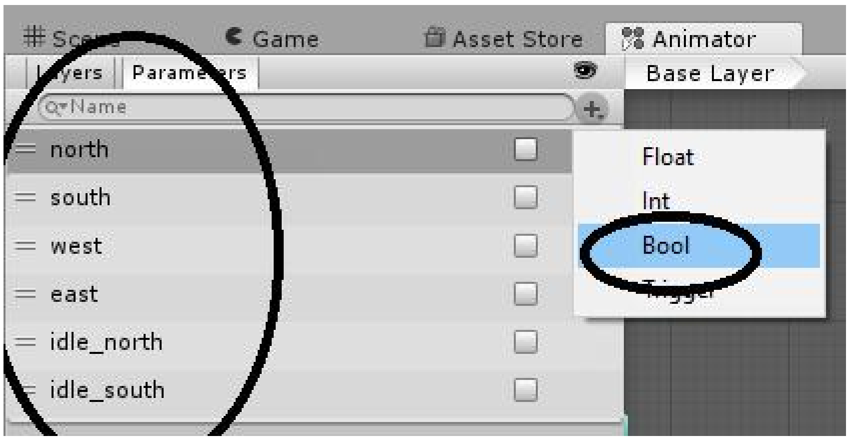
Warning: Exporting lists will NOT export sub-lists (like "quests to get") or objects. Only text, integers and boolean options will be exported to XML. Please, ensure that you have imported all necessary quests before further configuring nodes.

Player

Creating a player is extremely easy in P.A.C. Engine. First of all, create any gameObject on the scene outside of canvas, with a sprite renderer attached to it. Secondly, attach any collider to the player and adjust the size of it. Lastly, tag the object as "Player".

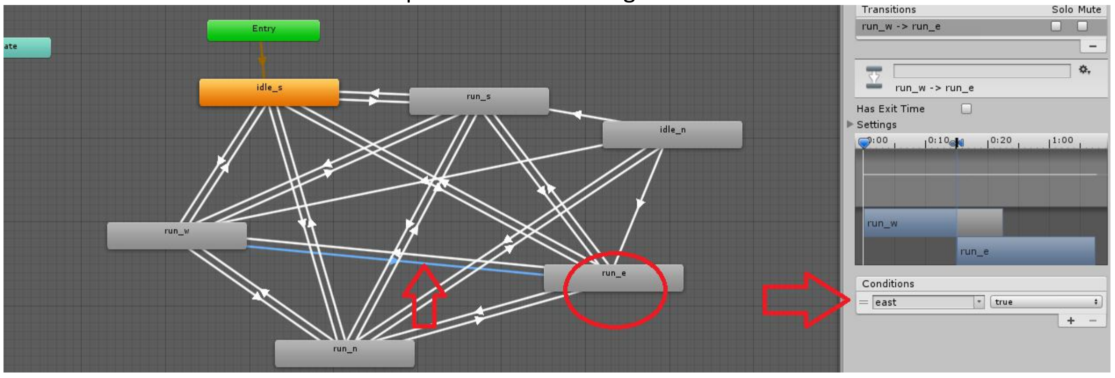
Player Animation:

Creating player animation and setting up the appropriate variables is a little bit more tricky. First and foremost, follow this tutorial on the unity official page to create your animations. Once you feel familiar with animations, head to your animations tab and create the following booleans.



Afterwards, make sure that each transition between animation has one of the latter parameters are requirement. This way, when "north" is true, run_n (run north) should play. If in your version of the game you have different animations, please modify the movement.cs script and alter/add any animations of your choosing.

Your animation screen should at this point look something like this:



Tags / layers

Although scarcely used, tags and layers are essential when creating any game. In P.A.C. Engine, the tags / layers used are as follows:

Tags:
Player: The tag used to mark the Player object.
Ground: The tag used to mark objects that the player is supposed to be able to walk on.
MainCamera: In P.A.C. Engine there should be only one main camera tagged accordingly.
Entity: The tag used to mark entities.
Item: The tag used to mark items.
LevelManager: This tag should only appear once per level on the levelManager object.
SpeechHolder: This tag should be attached to the speechHolder object under the canvas (since all text is being spawned under that object).
PlayerTextPoint: The gameObject to serve as the spawn coordinates for player's speech bubble.

Layers:
UI: Only canvas object should exist under the UI layer.
CollisionLayer: Any object that the player is supposed to be able to collide with should exist under the collision layer.
MovementCatcher: The original movement catcher object exists under the canvas and allows making animation direction decisions more easily. All objects located under the latter one should exist under the layer in question.

Stages

In the majority of my assets I use stages to allow character interaction progression. A stage in essentially an indicator of what should the entity do when the player interacts with it. After certain interactions the stage might change, changing how the entity will act the next time it is interacted with.

MovementCatcher

Movement catcher is a gameObject is located under canvas which is hosting an array of interactable UI elements designed to detect animation direction. Each one of the object under the one in question is a gameObject with an animationDirector script attached to it. The latter script takes as input a direction abbreviation such as “nw” for north west. When clicked, the script let’s movement.cs know in which direction the player is supposedly moving in order for it to play the correct animation. Feel free to scale any of these objects as you like.

Level Manager

The levelManager is a gameObject which is supposed to exists on every scene. The latter allows the configuration of various variables in a unique fashion for each level. The default adjustable values are as follows:

1) Step Sound Id: The id of the movement audioClip in the database.
2) Background Ambience & music ids: The ids of the starting background ambience sound and music in the database.
3) Play Background Music: Should any background music be played at start ?
4) Play Ambience: Should any ambience be played at start ?

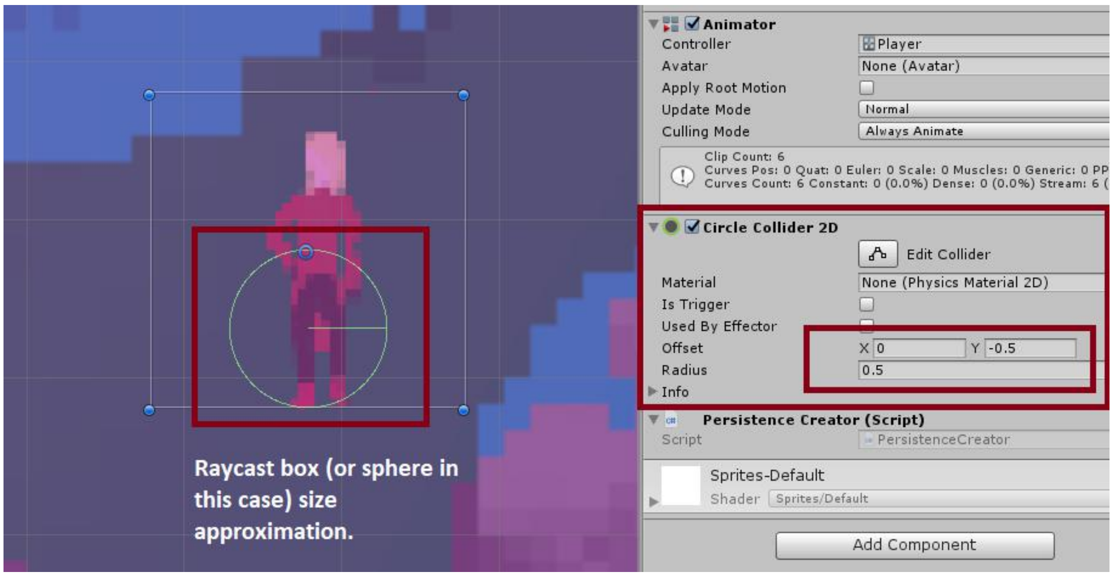
Movement (Under GameManager)

As mentioned before, this script is responsible for character movement and animation. The editable values are as follows:

1) Movement refresh speed: When moving, the player constantly checks whether there are any obstacles in the way along with other checks. Since the latter checks are executed in a coroutine, a delay is required in order to avoid crashes. The value in question represents that delay.
2) Play step sound: Should any step sound be played when our character is moving ? - Note: The the sound is declared in the LevelManager.
3) Step Clip Speed: Delay between each “play” of the above audioclip.
4) Animation Direction: The default animation direction. The value will depend on how you have configured movement.cs and your player animations.
5) Player can move: The player will only be able to move if this value is set to true.

6) Raycast spawn adjustment, castDistance & Size:
Movement in P.A.C. Engine is heavily based on raycasting, or, more specifically, line casting. As the player moves towards certain point (or attempts to do so), a line is cast between the player at the point in question.If the line hits a collider and the collider is too close to the player, the player stops moving.

In order for the above to work, the box is given a certain **size** and **distance** to travel. The distance is important since the box is cast continuously and expires (disappears) as soon as it travels that distance. In other words, a number of boxes is generated and cast during the player's movement. If the distance set for the generated box to travel is too short, the player will probably "collide" with the object before stopping. On the other hand, if the distance is too long, the player will stop too early and the person behind the screen will probably be confused as to why the character suddenly stopped before reaching an obstacle. Lastly, the **rayCastSpawnAdjustment** vector is a pair of x and y coordinates which are added to the original spawn position of the raycast box (the centre of the player gameObject). For humanoids, I highly recommend creating a small box at the very "feet" of the player character. This way any movements and collisions will feel more natural. You can approximate the size and adjustments needed for your box by creating a simple 2d box collider on your character, making the these adjustments and simply copying the values.



Animator
Controller: Player
Avatar: None (Avatar)
Apply Root Motion: ☐
Update Mode: Normal
Culling Mode: Always Animate

```
Clip Count: 6
Curves Pos: 0 Quat: 0 Euler: 0 Scale: 0 Muscles: 0 Generic: 0 PP
Curves Count: 6 Constant: 0 (0.0%) Dense: 0 (0.0%) Stream: 6 (
```

Circle Collider 2D
Edit Collider
Material: None (Physics Material 2D)
Is Trigger: ☐
Used By Effector: ☐
Offset: X 0 Y -0.5
Radius: 0.5
Info

Persistence Creator (Script)
Script: PersistenceCreator

Sprites-Default
Shader: Sprites/Default

Add Component

Note: **The actual box collider on your character does not matte**r. However, if you are planning to register on player clicks (which you can already do via the ClickManager.cs), the character collider should preferably be the same size as the character.

Note2: The type of the collider and raycast does not matter either. By default a sphere collider is used on the character and a raycast box is generated to locate obstacles, but feel free to change the latter two to anything suiting your project.

Cursor Changer (Under GameManager)

This script allows to specify the cursor to be displayed during (or outside of) interactions. Each cursor in the cursors list must have a unique id and a texture (image marked as "cursor" in its settings). By default there are two states of cursors, **normal** and **highlight**. You can specify (by id) which cursor should be used in each state. States are switched by calling the appropriate function from scripts such as "entity.cs" and "item.cs".

Note: All UI elements must have 2d box colliders attached in order for the script to work properly.

Set up:

In P.A.C. Engine, persistent objects play a vital role in the proper functioning of the game. Thus, these objects should be always initiated at the start of the game (and only then). No levels in the game, except the "initiation" level, should save any persistent objects on them. Otherwise, you may end up with duplicated when loading levels. The initiation (or initiator) level is essentially a level with all persistent objects and one additional object which is designed to load a specified level once all persistent objects have been fully loaded.

Once have set up all your persistent objects on the initiation level, head to the initiationManager object and modify the player spawn coordinates and load level id, to load the level and spawn the player in the proper location. As mentioned before, the level to be loaded should be a level consisting only of on-scene objects (no canvas, camera or persistent objects are to be present).

You may notice that under the canvas of the “initiator” level, there is an object called “SceneBlocker”. This object is simply present to hide the UI while the player is on this level. The objectDestroyer script, present on the object, will destroy the object once another level is loaded, and thus revealing the UI and scene to the player.

General notes

1) Build settings: All scenes used in your game must be added to build settings (File -> Build settings). Additionally, all scenes (excluding scenes such as the main menu which should not contain any of the persistent objects) and their respective build settings indexes must exist in the “locations” section of the database.

2) Customizing the appearance of Entity.cs: Due to the restrictions unity imposes, the file responsible for the visual side of Entity.cs is located in the “Editor” folder, under the name “EntityEditor.cs”. If you are interested in modifying the visual appearance of the script in the inspector, please follow this link to read more on the topic.

Conclusion

More features will be added to this asset over time. This documentation will be updated in future increments of the asset, or/and on demand.