Godot Engine: Technical Report

Matthew Holman - Department of Computer Science

Abstract

***Abstract*—This document presents a technical report which outlines the use of the Godot game engine in writing a paper about inter-agent communication.**

***Keywords*—inter-agent communication, technical report**

Introduction

In the course of an independent study class at the University of Idaho, the goal of which was to produce a publishable research paper, it was necessary to create a game-simulation that would test the aspect of AI programming that the class focused on. The focus of the class was 'inter-agent communication,' and the purpose of it was to test the effects of AI agents communicating with one another when they are given a simple resource collection task. These results were compared to the results of tests when the agents are *not* communicating.

To this end, the Godot game engine was selected to create the necessary simulation. This engine was chosen because, according to the engine's homepage, it is both "completely free and open-source under the very permissive MIT license," and is "easy to program" [1].

Working With Godot

Learning any new technology can sometimes be difficult. To aid in learning how to work with the Godot engine, and how to program in GDScript, the author first completed several tutorials. These tutorials came primarily from either GDQuest on YouTube [2], or from a video course purchased from Packt Publishing [3]. Completing several of these tutorials led the author to a better understanding of how the Godot engine functioned, and what kind of workflow would be appropriate.

Installing

There was no installation of the Godot engine *per se*, meaning that there was no msi or exe installer. Rather, 'installation' was as simple as downloading the appropriate version of Godot from the website, unzipping the folder that was downloaded, and changing the name of the executable to *Godot.exe*. Double clicking that executable automatically launches the Godot engine/ editor with no further setup required.

GDScript

While the Godot engine has a variety of language options, such as "GDScript, Visual Scripting, C#, and C++" [4] the NPC_1 simulation was programmed with the *GDScript* language, which is the default in the editor. According to the online documentation, GDScript is specifically meant to be similar to the Python programming language [5].

There was one difference between the syntax in GDScript and syntax Python that was unexpected, and which caused a number of delays in the codeing. This difference was in regards to the 'not' keyword. In Python, an expected use of 'not' would be as follows:

```
loc not in self.crystal_locations
```

However, in GDScript, the 'not' keyword is to be used as follows:

```
not loc in self.crystal_locations
```

According to user akien-mga in a response to an 'issue' posting on the Godot GitHub page "The not in operator exists in Python but indeed not in GDScript" [6].

Fortunately, this was the only notable challenge in using the GDScript language.

Command Line Arguments

Running the various tests for the research paper required a level of automation that would allow for different variables in the simulation to be changed from the command line. A version of the Godot engine, previous to version 3.2.1, was very challenging in terms of accepting command line input. Per the author's memory, this was because all the command line strings needed to be prepended with a leading 'dash' character (e.g. '-1' for the non-negative number '1'). Obviously, this was unacceptable. Fortunately, as of Godot version 3.2.1, the engine takes command line arguments as one would expect (e.g. '-1' for negative one, and '1' for positive one).

Command line arguments are easily accessible via a call to 'OS' as follows:

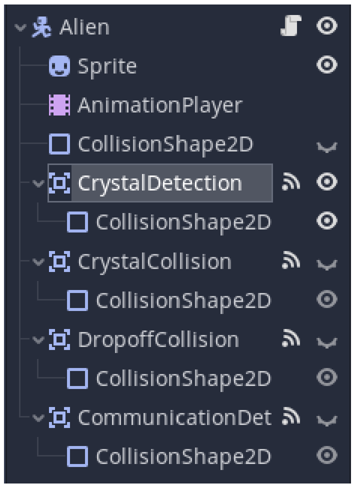
```
argv = OS.get_cmdline_args()
argc = len(argv)
```

The command line arguments, stored here in a list assigned to 'argv' can be accessed as below:

```
map_bounds.x = int(argv[0])
```

Scene Tree

Every project made in Godot exists in a tree structure. There is a central node, representing the game, and all the other nodes used in the game, also known as *scenes*, such as the map, aliens, crystals, timer, etc. are descended from that node. Individual *scenes* are also trees themselves. For example, the 'Alien' scene, represented in Figure 1, is the tree of nodes including the *Sprite* node, a *CollisionShape2D* node, and a variety of *Area2D* nodes with their attached *CollisionShape2D* nodes.



```
Alien
  Sprite
  AnimationPlayer
  CollisionShape2D
  CrystalDetection
    CollisionShape2D
  CrystalCollision
    CollisionShape2D
  DropoffCollision
    CollisionShape2D
  CommunicationDet
    CollisionShape2D
```

Figure 1: Alien Scene Tree

Some nodes, such as the *Area2D* node *require* child nodes, such as the *CollisionShape2D*. These 'collision shapes' can be set by the programmer to use a variety of form factors such as circles and squares. The programmer can set the sizes and positions of these, and other, items in the 'scene construction view,' next to the scene tree interface.

Scene Construction

The scene construction view, as seen in Figure 2, allows the programmer to move and position the various sprites, collision shapes, and various other nodes relative to a central location. The central location that the scene is centered around is indicated by the intersection of yellow and red colored lines.

Positioning aspects of a scene is easy to do, as there is a useful 'snap to grid' feature. However, although various aspects of a scene may be properly placed, functionality

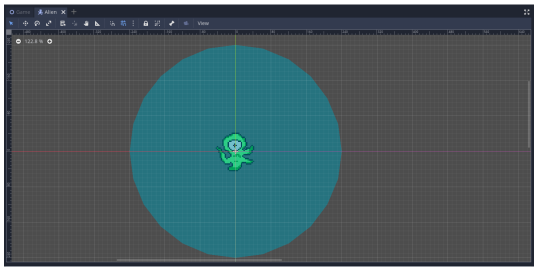


Figure 2: Alien Scene Construction

must still be added to these parts of the scene to make them useful.

Signals

The primary way that the Godot engine adds functionality to a scene is through the use of *signals*. Almost all the various nodes available to the Godot engine have a number of signals that can be activated. For example, the tree in Figure 1 has a selected *Area2D* node, renamed 'CrystalDetection,' with a signal attached to it. In Figure 3, one can observe that the red 'area_entered' signal at the top of the Figure has a green child, named '_on_CrystalDetection_area_entered().' This is the name of the function that is called whenever an applicable CollisionShape2D object, which is attached to an Area2D object, enters the collision shape attached to the original Area2D object with the signal attached to it.

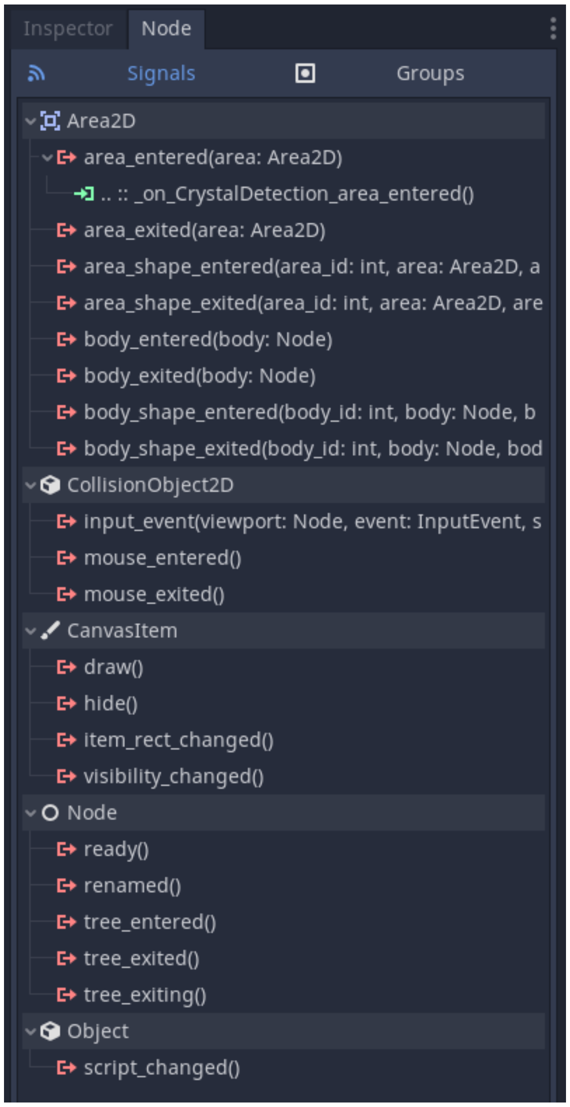
In the case of the activating the signal described above, the Godot engine knows which items activate the signal, and which do not, through the use of *collision layers*. For example, the scene that represents a crystal for the alien to collect is set to *exist* on the 'Gems' layer, as seen in Figure 4.

Once a scene, such as the crystal mentioned above, is set to 'exist' on a specific layer, other scenes can register interactions with that scene. In this case, the alien scene can interact with items existing in the 'Gems' layer, so long as the alien scene's *mask* is set appropriately, as seen in Figure 5.

Every time a item existing on the 'Gems' layer touches the CollisionShape2D of another entity, so long as that entity's mask set to the 'Gems' layer, any associated signals will be executed. In this case, when a crystal enters an alien's vision detection radius, the '_on_CrystalDetection_area_entered()' function will be executed. Please see Appendix B - Code Example for an example of this code.

Path Finding

One of the largest challenges in implementing the simulation was in making the path finding work properly.

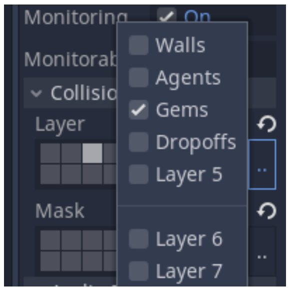


Inspector | Node

Signals | Groups

- Area2D
  - area_entered(area: Area2D)
    - .. :: _on_CrystalDetection_area_entered()
  - area_exited(area: Area2D)
  - area_shape_entered(area_id: int, area: Area2D, a
  - area_shape_exited(area_id: int, area: Area2D, are
  - body_entered(body: Node)
  - body_exited(body: Node)
  - body_shape_entered(body_id: int, body: Node, b
  - body_shape_exited(body_id: int, body: Node, bod
- CollisionObject2D
  - input_event(viewport: Node, event: InputEvent, s
  - mouse_entered()
  - mouse_exited()
- CanvasItem
  - draw()
  - hide()
  - item_rect_changed()
  - visibility_changed()
- Node
  - ready()
  - renamed()
  - tree_entered()
  - tree_exited()
  - tree_exiting()
- Object
  - script_changed()

Figure 3: Alien Signals



Monitoring ✓ On

Monitorab

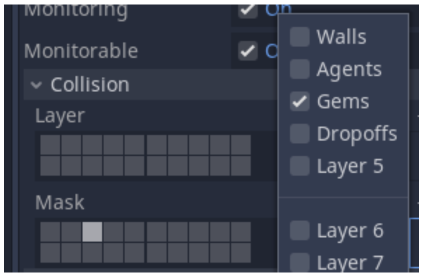
Collisio

Layer

Mask

- Walls
- Agents
- ✓ Gems
- Dropoffs
- Layer 5
- Layer 6
- Layer 7

Figure 4: Crystal Exists on the Gems Layer



Monitoring ✓ On

Monitorable ✓ O

Collision

Layer

Mask

- Walls
- Agents
- ✓ Gems
- Dropoffs
- Layer 5
- Layer 6
- Layer 7

Figure 5: Alien Set to Collide With Gem Layer

There were two solutions that were implemented. The first solution, seen in Figure 6, demonstrates the path finding solutions generated when using a 'navigation mesh.'

The Godot engine, ostensibly, allows the programmer to create a three-dimensional 'mesh' that is associated with the various tiles that go into making the map. Once the tiles are arranged into a map, the underlying mesh pieces are then connected to one another. An example mesh piece can be seen in Figure 7.

However, the author had difficulty getting this particular path finding solution to produce acceptable results. As such, the A* algorithm was used in place of the navigation mesh.

Fortunately, the Godot engine had a pre-built node that executes the A* algorithm [7]. However, using this node required that a directed graph be created in memory for the A* algorithm to traverse. Furthermore, the nodes in the directed graph needed to be connected to one another manually. This need for manual graph creation and connection allows for maps of any shape, configuration, or number of nodes. However, while this allows Godot's A* implementation to be very flexible, the amount of 'overhead' needed to get it to work properly did seem to the author to be a bit extraneous. Ultimately, the author found an MIT-licensed demo created by GDQuest [8], and was able to modify the code therein to work for the simulation under consideration. Figure 8 demonstrates the path finding solutions using A*, which are far more preferable in terms of 'making sense,' than the path demonstrated by the navigation mesh solution, as seen in Figure 6.

Overall Impression and Conclusion

Ultimately, the author found working with the Godot engine to be an acceptable experience. Although he did not work with too many of the engine's advanced features, such as 3D rendering, he did feel that he learned enough about the technology to be able to recommend it to others.

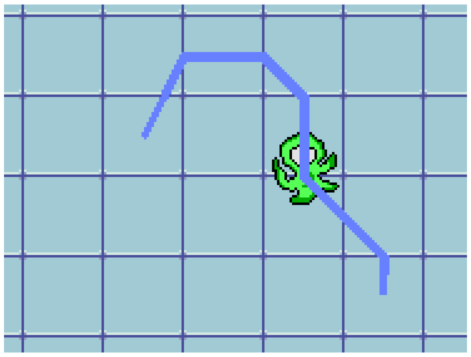


Figure 6: The First Path Finding Solution

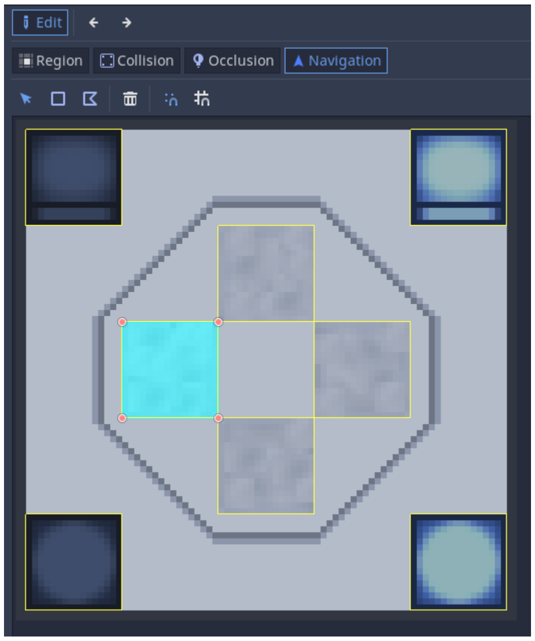


Figure 7: Navigation Mesh Piece

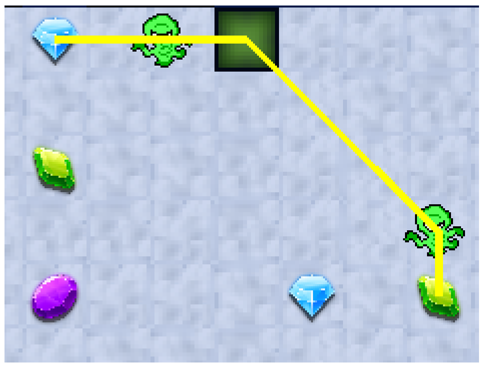


Figure 8: A* Path Finding

I. Appendix A - Links

Github

https://github.com/mholman5721/Agent_Communication

Godot

https://godotengine.org/

GDQuest

https://www.youtube.com/channel/UCxboW7x0jZqFdvMdCFKTMsQ

II. Appendix B - Code Example

```
func _on_CrystalDetection_area_entered(area: Area2D) -> void:
        var loc : Vector2 = tile_map.get_map_loc_from_pos(area.position.x,
           ↪ area.position.y)
        if GlobalValues.detecting_crystals and not loc in self.
           ↪ crystal_locations and not loc in self.crystal_location_blacklist
           ↪  and area.visible == true:
                self.crystal_locations.append(loc)
                print("## CRYSTAL DETECTION: _on_CrystalDetection_area_entered
                   ↪  - Agent: ", self.ID, " has found crystals at: ", self.
                   ↪ crystal_locations, " and will not re-detect those at: ",
                   ↪  self.crystal_location_blacklist, " unless through
                   ↪ communication.")
                Logger.info("## CRYSTAL DETECTION:
                   ↪ _on_CrystalDetection_area_entered - Agent: " + str(self.
                   ↪ ID) + " has found crystals at: " + str(self.
                   ↪ crystal_locations) + " and will not re-detect those at:
                   ↪ " + str(self.crystal_location_blacklist) + " unless
                   ↪ through communication.")
```

References

[1] J. Linietsky, A. Manzur, and contributors, “Godot Engine - Free and open source 2D and 3D game engine.” https://godotengine.org/. Accessed: 2020-05-17.

[2] GDQuest, “Become A Game Developer.” https://www.youtube.com/channel/UCxboW7x0jZqFdvMdCFKTMsQ. Accessed: 2020-05-17.

[3] C. Bradfield and D. W. Parker, “Ultimate Godot Game Developer Projects.” https://www.packtpub.com/game-development/ultimate-godot-game-developer-projects. Accessed: 2020-05-17.

[4] J. Linietsky, A. Manzur, and the Godot Engine community, “Frequently asked questions.” https://docs.godotengine.org/en/stable/about/faq.html. Accessed: 2020-05-16.

[5] J. Linietsky, A. Manzur, and the Godot Engine community, “GDScript basics.” https://docs.godotengine.org/en/latest/getting_started/scripting/gdscript/gdscript_basics.html. Accessed: 2020-05-16.

[6] akien-mga, “Parsing error when using ’not in’ #3677.” https://github.com/godotengine/godot/issues/3677. Accessed: 2020-05-17.

[7] J. Linietsky, A. Manzur, and the Godot Engine community, “AStar.” https://docs.godotengine.org/en/3.2/classes/class_astar.html. Accessed: 2020-05-16.

[8] N. Lovato and A. Franke, “pathfind_astar.gd.” https://github.com/GDQuest/godot-demos/blob/master/2018/03-30-astar-pathfinding/pathfind_astar.gd. Accessed: 2020-05-13.