SimplePath 1.11

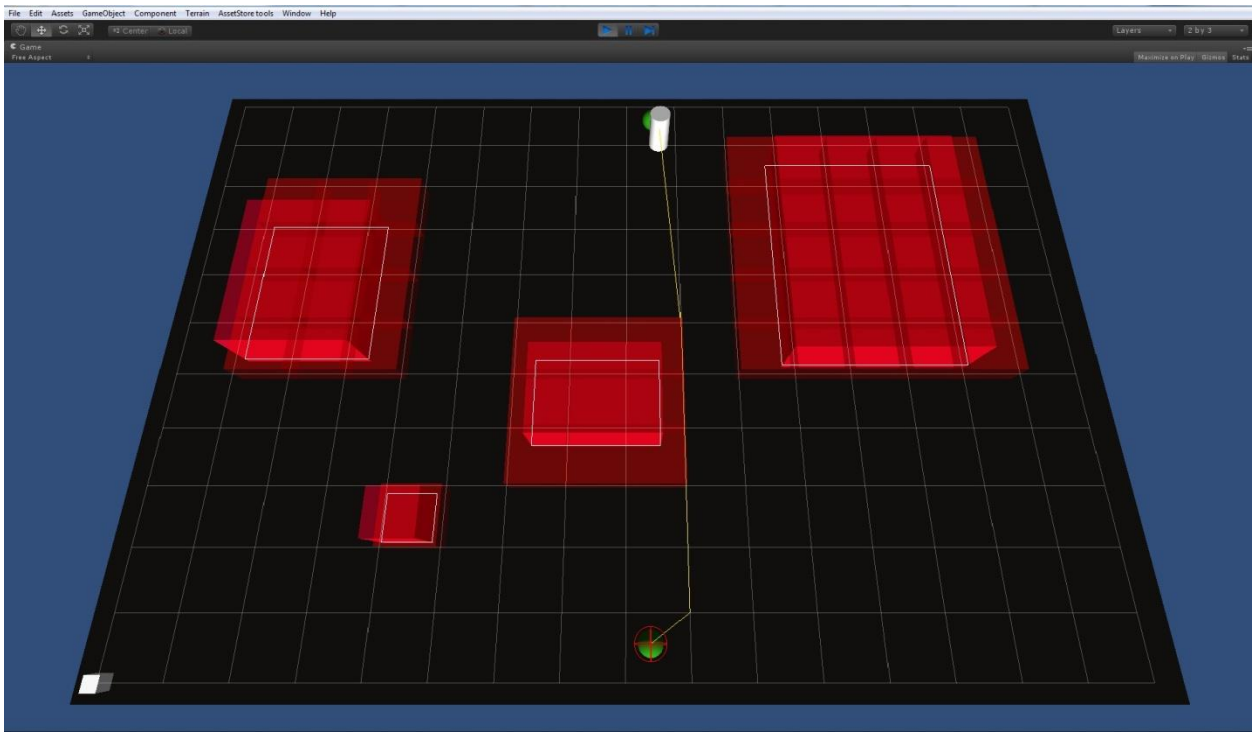
User's Manual

Contents

Introduction

The purpose of SimplePath is to provide game developers with a robust toolkit for pathfinding within the Unity game engine. Like most pathfinding software, SimplePath was designed for use with any game genre. SimplePath provides everything you need to get your AI navigating in game. While the software provides simple steering, and a grid terrain representation, the main focus is on robust pathfinding. The system is designed to allow you to easily plug-in your own steering system and terrain representation if you desire. In the following sections of this document, I will describe how you can utilize SimplePath for your own games.

Starting at the highest level, there are primarily three game structures of importance. These are the agent, the PathManager, and the terrain. These three entities define the pathfinding universe. The agent is the GameObject that moves around the world, the PathManager is responsible for servicing requests from many agents, and all of this takes place on the Terrain, where the agent is moving. The following screenshot of the "PatrolWithObstacles" example scene depicts these major components.



The agent is the white cylinder, the terrain is the white grid, and the PathManager has no physical representation. Additionally, there are several other notable elements depicted in the above figure. The yellow line is the path; the red wire-frame sphere is the goal; the green spheres are patrol nodes; the large red cubes are obstacles; the red area beneath each obstacle is the footprint of that obstacle; the large white rectangles represent the bounding box of each obstacle; and the black square is the floor.

Quick Start

If you're only interested in the bare minimum to getting things up and running, here's what you need to do. Place the following three GameObjects in your scene (SimplePath comes with prefabs for each of the following GameObjects).

- PathManager
- PathGridWithObstacles
- Agent_Wander

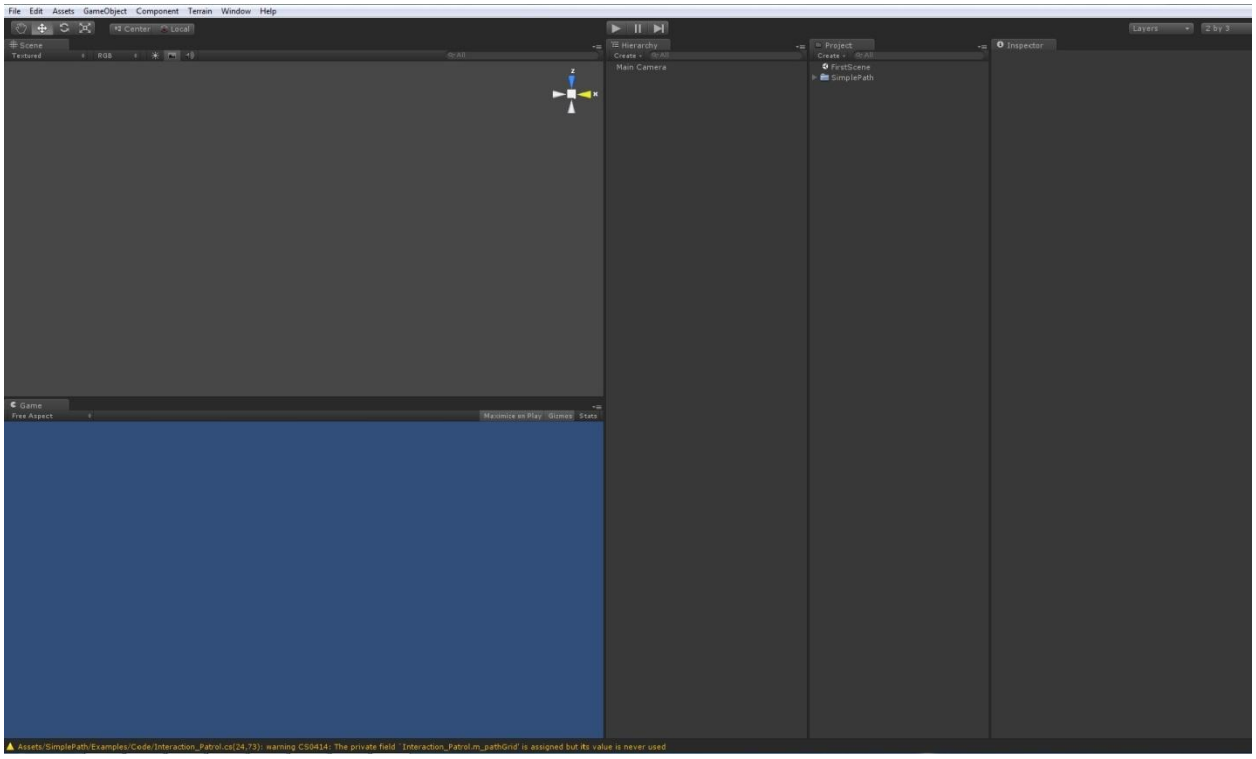
You'll then need to hookup the following variables in the Inspector window.

- Agent->PathAgentComponent->PathManager
- PathManager->PathTerrainComponent

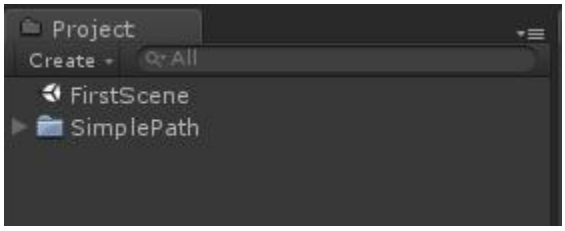
Press play, and the agent should randomly wander around the grid. Next, I will describe the step-by-step process to create the PathWithObstacles scene.

Tutorial - Creating Your First Scene

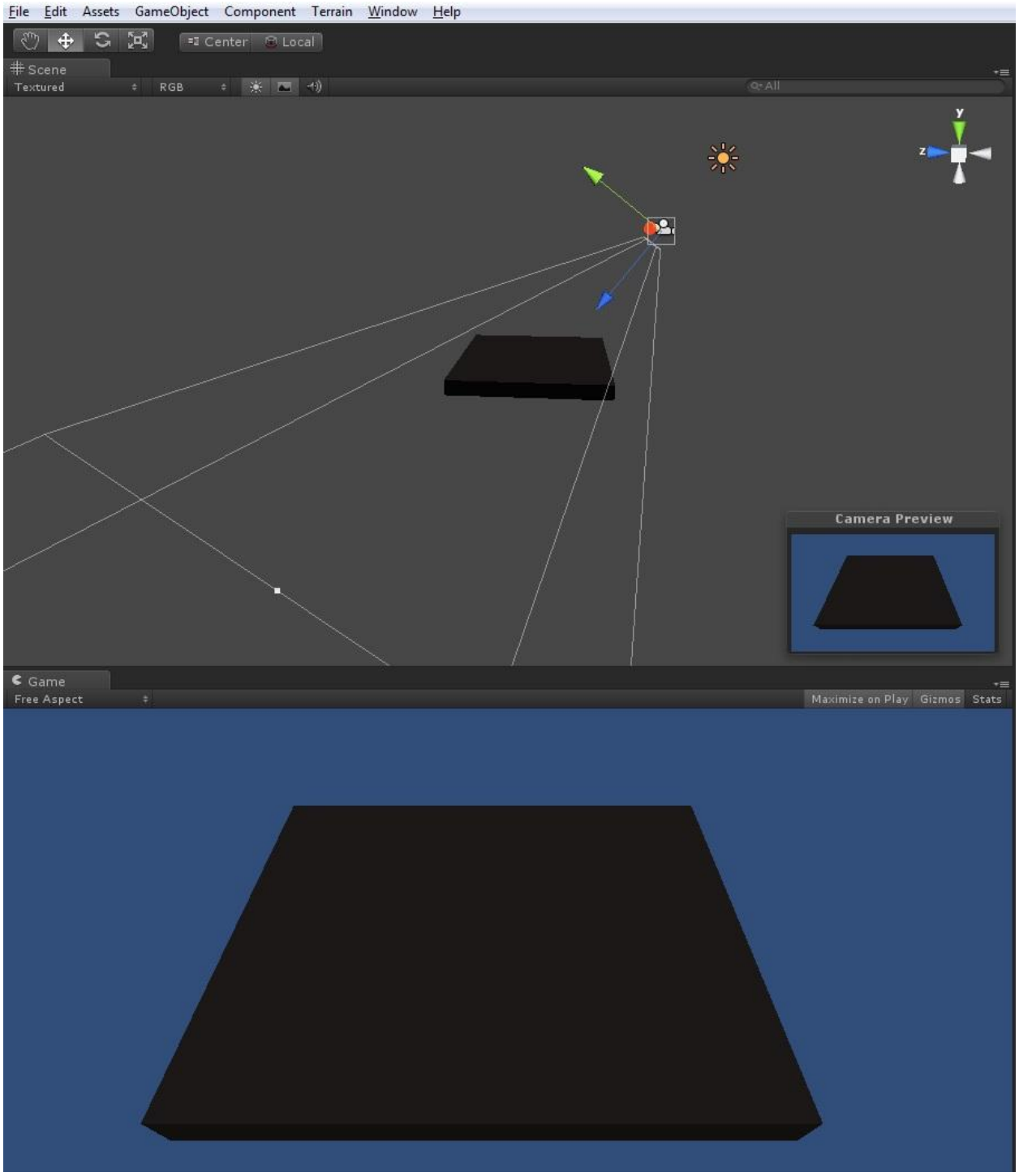
First, open up Unity, and create a new project. Then, import the "SimplePath" package by selecting Assets->Import Package->SimplePath. Now create a new scene, and name it "FirstScene." Your resulting project should look something like the image below.



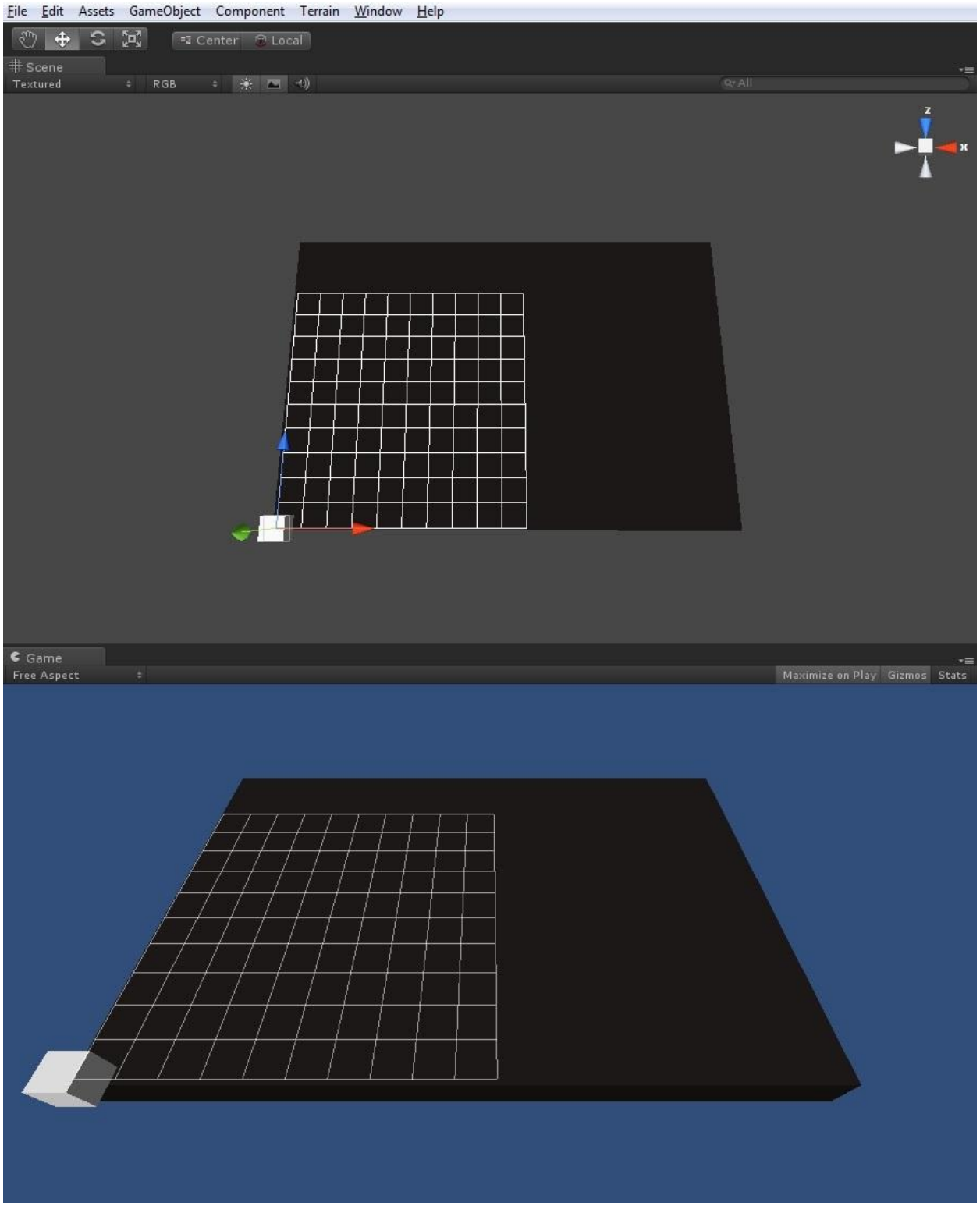
And the project window should look like this.



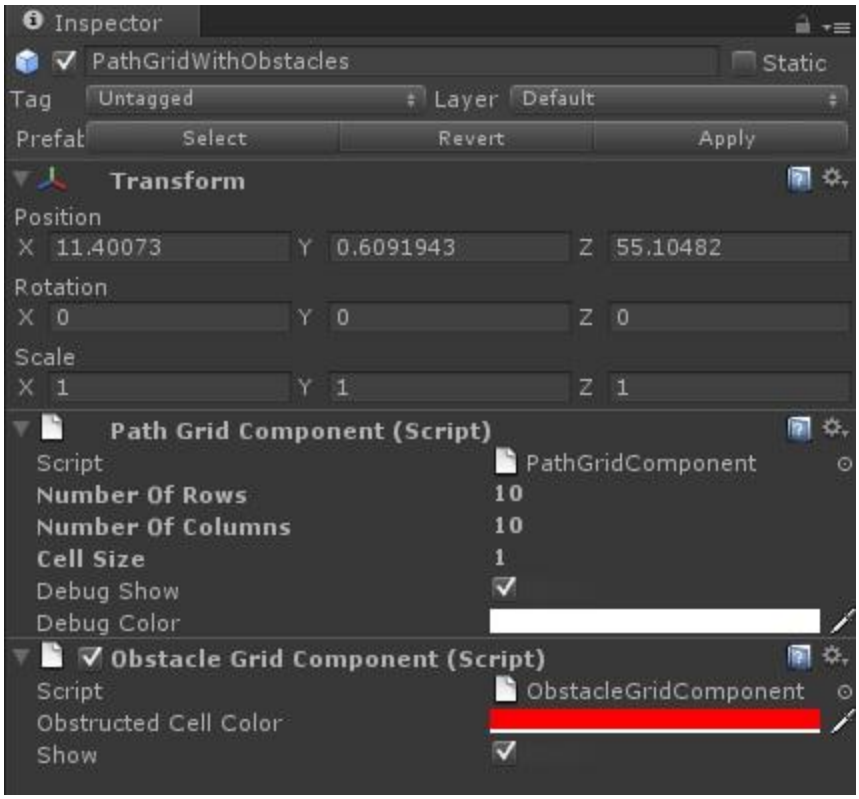
Now let's start adding some objects to the scene. Add a cube GameObject (GameObject->Create Other->Cube), and stretch it out a bit in the X and Z axis. This will serve as our floor, so name this object "Floor." Next, in the Mesh Renderer component, select Element 0 of Materials, and choose the Ground_Mat material. This should render your floor black, which will make it easier to see the next objects we're going to add. Drag the camera over toward the floor object we just created, and point it down the negative-Y axis. Finally, create a directional light, place it behind your camera, and your scene should look something like the following.



Next we will create the terrain. Locate the PathGridWithObstacles prefab located in the directory SimplePath->Main->Resources. Drag one of these objects into the scene, and you should see a white grid. Place the grid just above the floor's surface, and align the two objects. Your scene will look something like the following (don't worry if the proportions are different than the following screenshot, we will fix that next!)



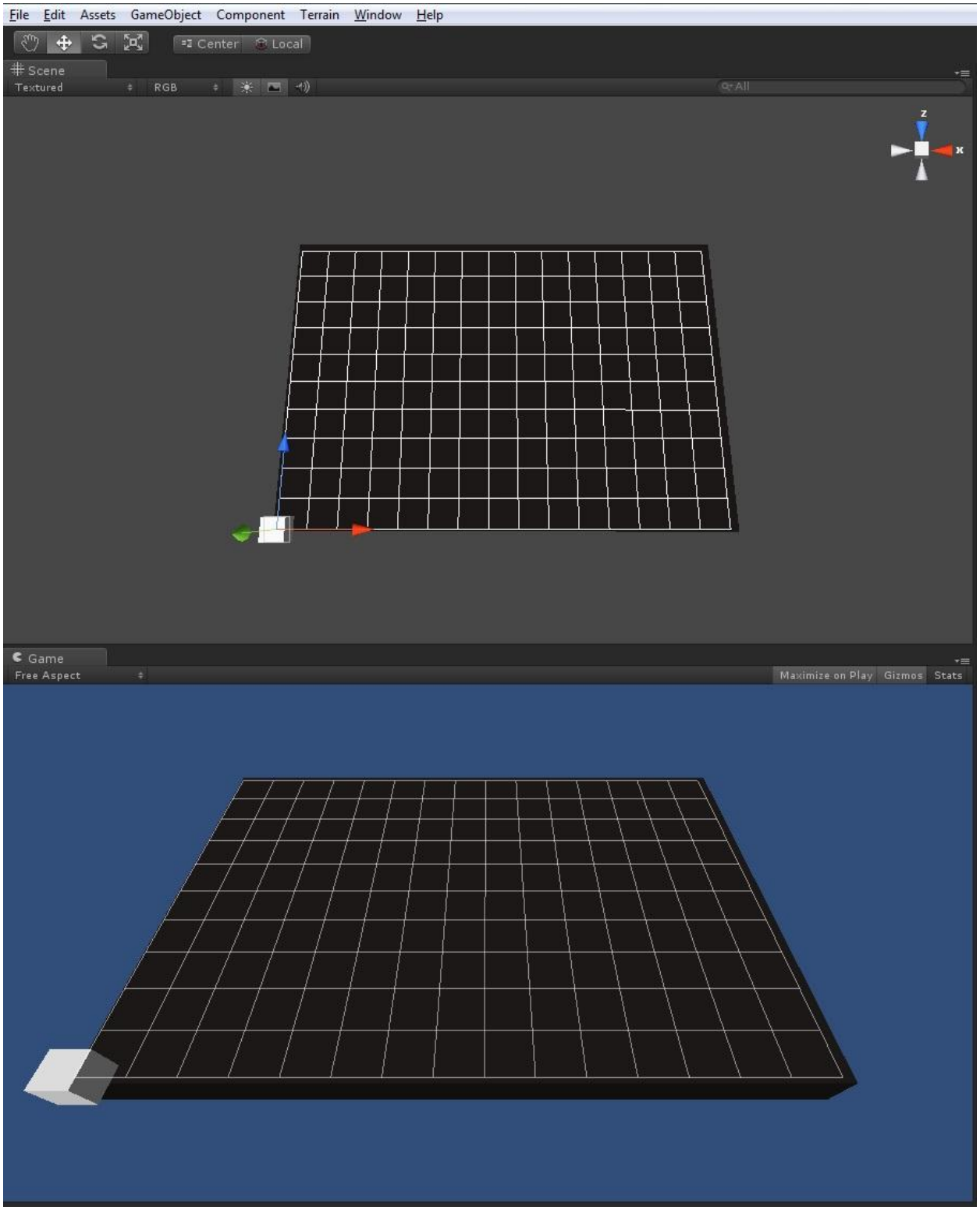
Now select the grid object you just created, and take a look at it in the Inspector. We're going to modify some of the grid properties, to make the navigation space match the floor.



Take a look at the Path Grid Component. The following variables are used to change the terrain.

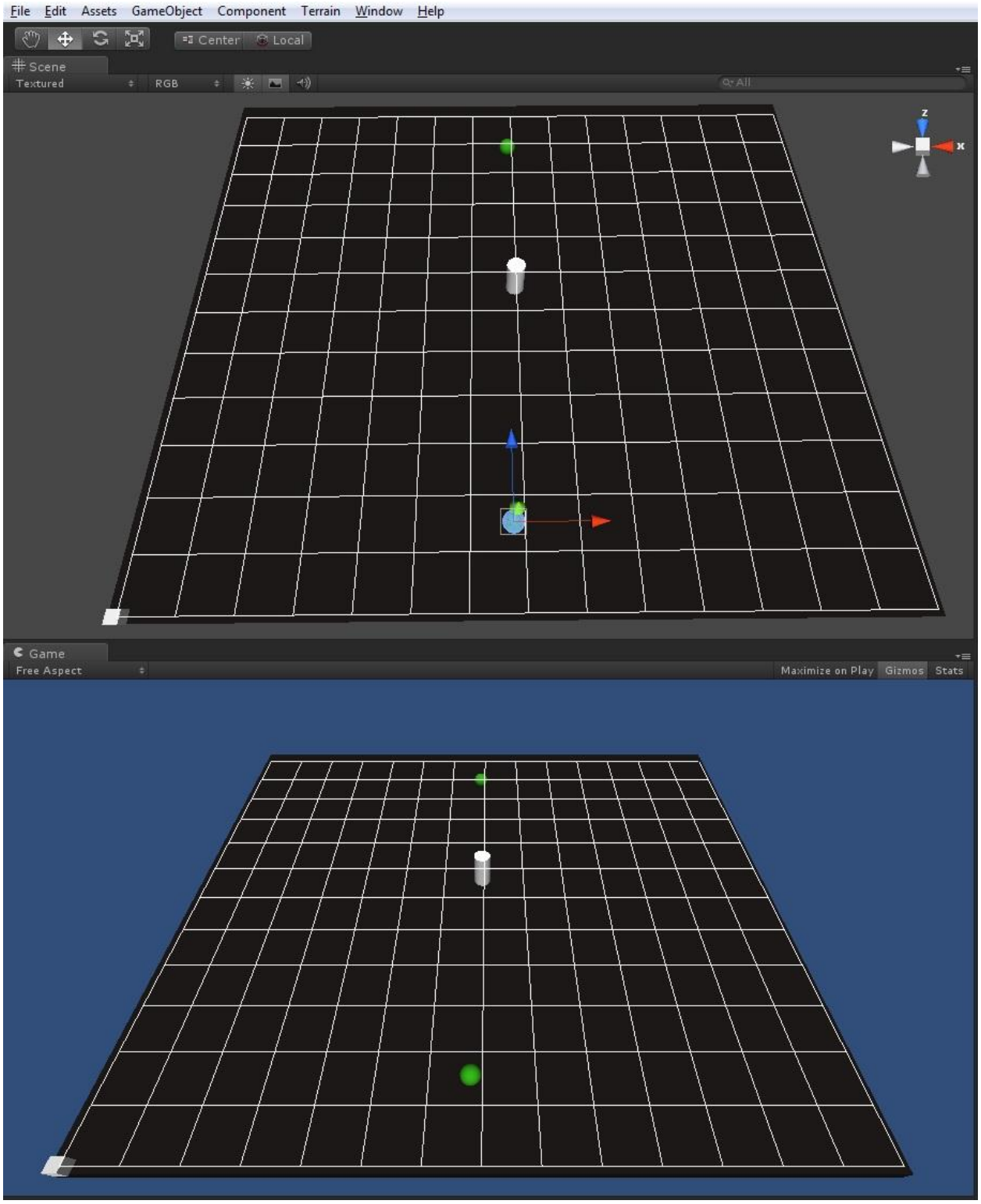
- Number of Rows
- Number of Columns
- Cell Size

Play around with these values, and notice how they affect the grid. Once you have a fair understanding, set the appropriate values that provide you with a grid and floor that are roughly the same dimensions, which should look like this.

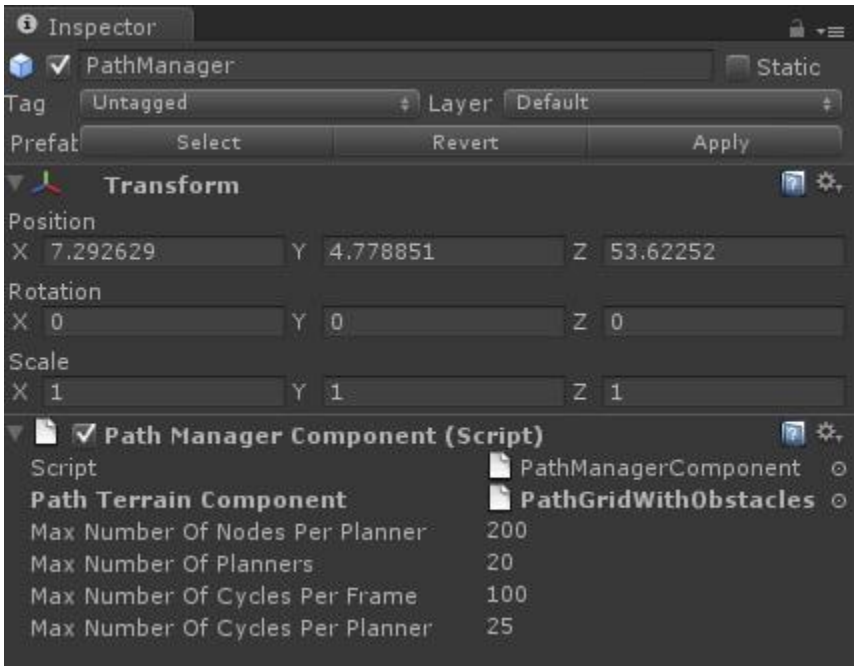


Now let's get an agent moving around in the world, between two patrol points. First, we'll need a PathManager object to solve all of the paths for us. Find this prefab located in SimplePath->Main->Resources. Next, we need the actual agent that moves around the world. Find the

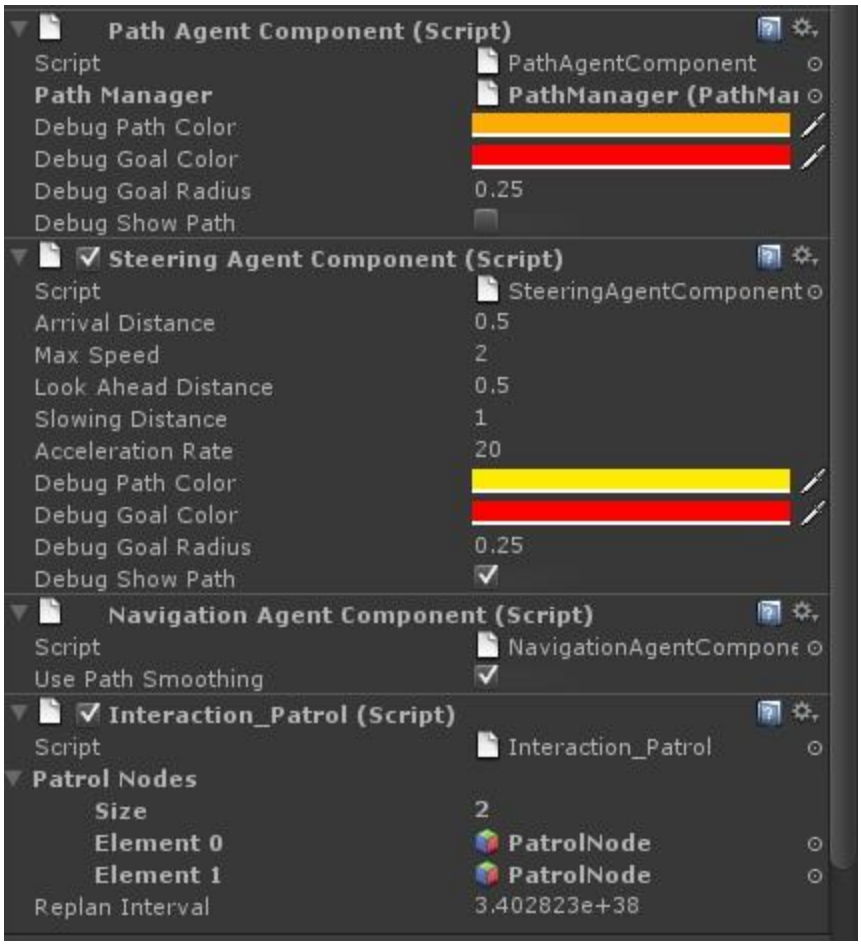
Actor_Patrol prefab located in the directory SimplePath->Examples->Resources->Prefabs. Drag one of these into the scene, and place him somewhere on the floor. Then, add two PatrolNode objects to move between. These are located in SimplePath->Examples->Resources->Prefabs. Again, place these objects somewhere on the floor. Your scene should appear as follows.



Now that we have all of our objects, we need to make a couple of connections, so that the objects know about each other. The PathManager needs to know about the terrain for which it is solving paths. Select the PathManager object from the Hierarchy window. Now select the Path Terrain Component from the Inspector window, and connect it to the PathGridWithObstacles.

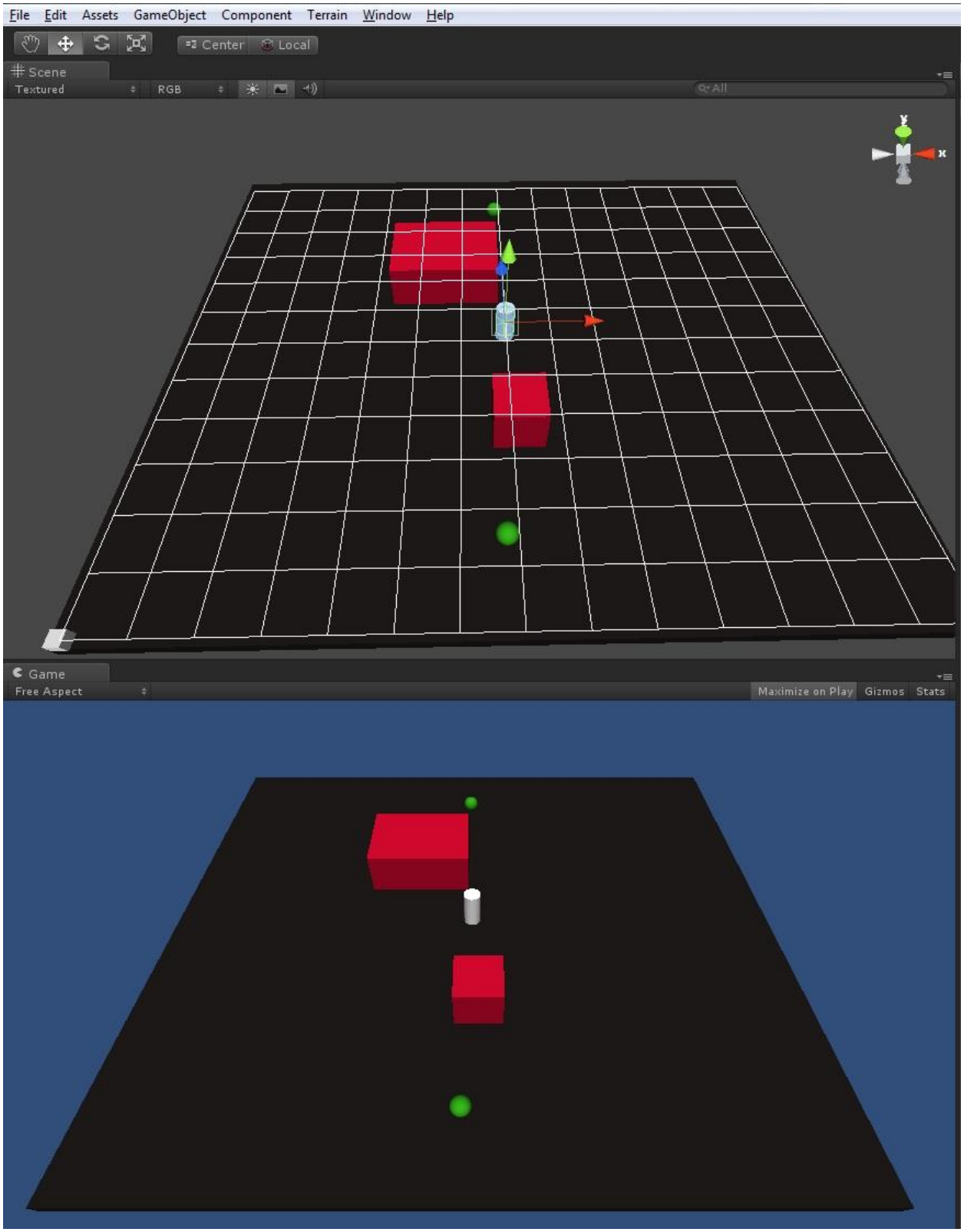


Next, we need to tell the agent about the PathManager that is solving his paths, and the PatrolNodes that he is supposed to move between. So select the Actor_Patrol from the Hierarchy window. Locate the Path Agent Component from the Inspector window, and find the Path Manager variable. Connect this to the PathManager object in the scene. Finally, locate the Interaction_Patrol script from the Inspector window, and look at the Patrol Nodes variable. Set this array to size 2, and connect the first element of the array to the first PatrolNode you created, and the second element to the second PatrolNode. Your inspector window should look something like the following image.



Now, you can press play, and watch your agent navigate between the two patrol points!

Next, let's make the agent's job a bit more complicated by adding some objects to the scene. Locate the Obstacle_Box prefab in SimplePath->Main->Resources, and drag a few of these into the scene. Place these objects on the floor, and size them to your liking (just make sure you don't cover up the PatrolNodes, or the agent). Press play again, and the agent should navigate around these objects. Your final scene should resemble the image below.



If you have gizmos turned on, you should notice a bunch of red rectangles below each obstacle. These rectangles represent the footprint of each obstacle. When the agent is navigating, he

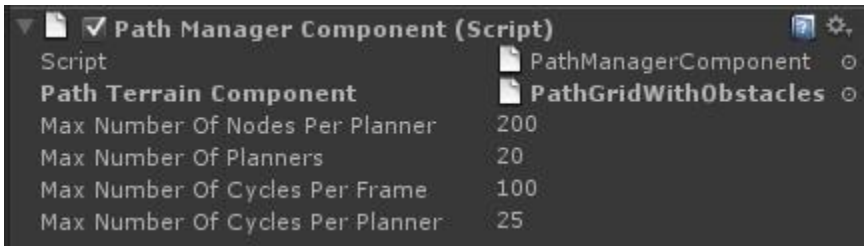
makes sure to avoid any terrain covered by a footprint. In the next section, I cover more details about each of the objects mentioned in this tutorial.

PathManager

This section describes the PathManager in detail, including all of the PathManager's tunable parameters, and the different ways the PathManager can be used. The PathManager is embodied by the PathManagerComponent, and is responsible for doing most of the work. Namely, it solves all of the path requests over time. The simplest task is for the PathManager to receive a request to find a path from point A to point B, and respond with the list of points from A to B, as soon as possible.

For most games, you will only need one PathManager, but SimplePath provides the flexibility to specify any number of PathManagers, where each PathManager can be responsible for managing a different set of agents (each agent specifies his PathManager in the Inspector window.)

All of the tunable variables that you find in the Inspector window for the PathManagerComponent, are there for controlling performance (except for the Path Terrain Component variable).



Here they are, described in detail.

- Path Terrain Component – Defines the terrain on which the PathManager is solving paths.
- Max Number of Nodes Per Planner – The size of the node pool for each planner. If a path search ever explores more nodes than this variable, then the search will fail. Increasing this variable will allow you to solve longer and more complicated paths, but it will also increase the amount of memory. The memory for each planner is pre-allocated (pooled), to avoid expensive Garbage Collection operations.
- Max Number of Planners – The size of the planner pool. Increasing this number will allow you to service more path requests at once, but it will also increase memory size. If the number of agents concurrently requesting paths exceeds this number, then several of those requests will not be serviced. For example, if there are 25 agents requesting paths, and this variable is set to 20, then 5 of those agents will not have their paths solved.

- Max Number of Cycles Per Frame – Each frame, the PathManager runs for this number of A* cycles. If you increase this number, performance will decrease, but paths will be solved in a fewer number of frames.
- Max Number of Cycles Per Planner – Each frame, a single planner can run for a maximum of this number of A* cycles.

The PathManager is the authority over all pathfinding, and is the single channel that all pathfinding must pass through. Next, I will describe more details about the terrain.

Terrain Representation

The terrain is the space where all navigation takes place. SimplePath comes with a grid terrain, but is structured to support any terrain type, whether it be a navigation mesh, waypoint graph, or grid. All of this starts with the PathTerrainComponent.

The PathTerrainComponent simply provides a place for the terrain to live; it just owns an IPathTerrain member variable. Every terrain component should inherit from this component. For example, the PathGridComponent inherits from PathTerrainComponent, and provides a wrapper for the PathGrid, as well as a place for all of the tunable grid parameters. SimplePath comes with three different Grid prefabs: PathGrid, PathGridWithObstacle, and PathGridWithObstaclesAndHeightmap. The difference being the obstacles and the heightmap support. The prefabs that support obstacles, allow for obstacles to be rasterized into the terrain, which allows the agents to pathfind around obstacles with a FootprintComponent. The PathGridWithObstaclesAndHeightmap also has a component that provides a heightmap; this is the HeightmapComponent_UnityTerrain component. This component specifies a Unity Terrain object as a heightmap, and therefore this component does not work with iPhone and Android (though Unity announced that Unity 3.4 will support the Unity Terrain object). You only need this component if you are planning on having uneven terrain in your scene. If you need uneven terrain on iPhone or Android, you can implement your own heightmap component, by inheriting from the IHeightmap interface. When a heightmap is present, the planner will be sure to account for the uneven terrain by changing the steering to steer over uneven slopes smoothly, and by changing the planner to weight the different heights in the terrain.

There is one other noteworthy variable, and it is on the ObstacleGridComponent. I am referring to the "Rasterize Every Frame" variable. When this is set to true, the dynamic obstacle grid will be raserized (updated) every frame. For certain games, this may be too expensive. If you need to better control the performance impact from the obstacle grid, you should uncheck this box, and manually call the Rasterize function only when necessary (ex: only when obstacles have moved). The rest of the terrain variables are self explanatory. Next I will explain how you can make your own terrain, via the IPathTerrain interface.

Create Your Own Terrain Representation

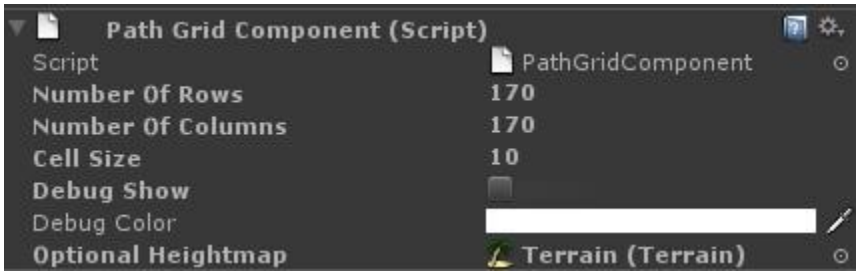
Every terrain type should inherit from this interface (ex: grid, nav mesh, waypoint graph). If you want to use your own terrain representation, there's a few things that you'll need to do.

1. Create your terrain data structure, and inherit from IPathTerrain. Use PathGrid as an example.
2. Create a component for your terrain, and inherit from PathTerrainComponent. Use the PathGridComponent as an example.
3. Attach your component to a GameObject. Use the PathGrid prefab as an example.

And that's it. Everything else should be setup the same, just as if you were using a grid terrain representation. One thing that might be different, however, is obstacle avoidance. Grids are particularly good at dynamic obstacle avoidance, since we rasterize the obstacles into the grid (via the FootprintComponent). If you decide to use a waypoint or nav mesh terrain representation, you may have to employ more complicated steering, or local planning, as well.

Uneven Terrain

The most significant addition in SimplePath 1.1 is the ability to navigate across uneven terrain. Supporting uneven terrain requires you to first create a Unity Terrain object, and then link it to the PathTerrain object.



After this value is set, the agents should successfully navigate across your Unity Terrain object. SimplePath will only use this object as a heightmap, and we only use the SampleHeight function. This function is called in one location, as shown in the code snippet below.

```
public float GetTerrainHeight(Vector3 position)
{
    if ( m_heightmap == null )
    {
        return Origin.y;
    }
    else
    {
        return m_heightmap.SampleHeight(position);
    }
}
```

You will likely want your agents to stick to the floor, and look realistic when moving across all the various bumps in the terrain. To control this, you can modify several variables. First, consider the variables on the SteeringAgentComponent. The max speed, acceleration rate, and gravitational acceleration rate all affect how the agent looks when he moves. If you want the agent to stick to the floor more, then you can increase the gravitational acceleration rate. But keep in mind that there needs to be a balance between the gravitational acceleration rate, and the acceleration rate. The acceleration rate only affects the XZ acceleration of the agent, while the gravitational acceleration rate only affects the Y acceleration. The other value to consider, is the mass of the agent, which is stored on the rigidbody. The downward force is computed as follows.

$$F_{gravity} = M_{agent} * A_{gravity}$$

In other words, SimplePath applies a downward force to the agent that is equivalent to the mass of the agent multiplied by the gravitational acceleration rate variable. Next I will discuss all of the different components on the agent.

Agent

The agent is the GameObject that physically moves around the world. Typically, this will be one of the enemies in your game. SimplePath comes with three example agent prefabs, which are the Agent_Wander, Agent_Chase, and Agent_Patrol, all located in SimplePath->Examples->Resources->Prefabs. The only difference between each of these agents is that they each have a different interaction script. Each agent also has a Path Agent Component, Steering Agent Component, and Navigation Agent Component. Next, I will describe each of these components, starting with the interaction script.

Interaction Component

The interaction script controls the agent's behavior. The navigation requests originate in this component. For your game, you might call this the behavior, the control state, or something else. The interaction script is there to show how you can plug-in SimplePath to your AI decision logic

(behavior tree, GOAP, FSM, etc.) In a commercial game, I would not recommend tying a single interaction script to an agent. Instead, you should attach some sort of InteractionComponent, which is responsible for running any type of interaction on the agent, where the interaction may be pushed onto the InteractionComponent from some BrainComponent that is making the decisions about which interaction to run.

Navigation Component

The navigation component controls navigation requests. Attach this component to any GameObject, and it will be able to navigate around the world. Any GameObject with a NavigationAgentComponent should also have a RigidBody, SteeringAgentComponent, and PathAgentComponent. This component has a "Use Path Smoothing" Inspector variable, which determines if it generates smooth or rough paths. By default, smooth paths are generated. If you want very blocky paths, that only go between the centers of the grid cells, then set this value to false (this behavior may be desirable for certain turn-based games). Note that pathsmoothing should work for any terrain representation.

Path Component

The path component is the agent's interface for pathfinding. Any GameObject with a PathAgentComponent can generate path requests. This component has a "Path Manager" Inspector variable, which determines which PathManagerComponent is in control of servicing the path requests for this PathAgentComponent. Typically, there will only be one PathManager in the entire game, but you have the flexibility to create multiple PathManager objects, and have each manager control different sets of agents based on this Inspector variable (this behavior may be desirable if you want to separate path requests for flying enemies and ground enemies).

Steering Component

The steering component takes in a list of points, and moves the agent along those points. In other words, it takes the output of the PathAgentComponent (the path solution), and steers the agent along that path. SimplePath provides you with a very simple steering solution. If you want something more complex, you should hook your steering system into this component. There are quite a few steering parameters, so I will go over each one in detail.

- Arrival Distance – when the agent is within the arrival distance of their destination, the navigation request will be complete, and the agent will stop moving. Imagine this as a sphere around the destination position. Once the agent gets inside this sphere, he stops.
- Max Speed – the fastest speed at which the agent can move.
- Look Ahead Distance – how far the agent looks ahead on his path. If this value is very small, the agent will stick tightly to his path. If it is very large, then he will follow his path more loosely. however, if you make this value too large or too small, the agent may get stuck.
- Slowing Distance – how far the agent must be from his destination before he begins to slow down. Set this value to 0 if you don't want the agent to slow down near his destination.
- Acceleration Rate – how fast the agent accelerates, in meters per second squared, in XZ.
- Gravitational Acceleration Rate – how fast the agent accelerates, in meters per second squared, in negative Y.

Creating Your Own Agent

SimplePath comes with three agent prefabs, but for most games, you will likely be creating your own agent. You can create agents that just solve paths (and don't actually steer along them), or you can create agents that will solve paths, and steer along them. The latter case is more common, and so I will describe a rough guideline for creating your own fully functional agent (your setup may slightly differ, depending on your game).

1. Select the GameObject that you want to turn into an agent.
2. Add the NavigationAgentComponent.
3. In the Inspector window, inside the RigidBody component, find the Constraints->Freeze Rotation variable. Make sure the X Y and Z boxes are checked. Also on this component, make sure the Use Gravity box is unchecked.
4. In the PathAgentComponent, hookup the Path Manager variable to a PathManagerComponent that is already in the scene.

After completing all of these steps, that agent should be able to move around the world. To get him to move from code, you can call MoveToPosition or MoveToGameObject on the NavigationAgentComponent, and wait for the OnNavigationRequestSucceeded or OnNavigationRequestFailed message to be passed to the GameObject. Or, you can attach one of the Interaction scripts to this GameObject (ex: Interaction_Wander).

Advanced Customization

There are several other aspects of planning that you can customize, which may be attractive to more advanced users. Next, I will briefly discuss the customization mechanisms that SimplePath offers, which have not already been mentioned in previous sections.

First, notice that the code decouples planning from the terrain. If you want to create a more general purpose planner (ex: for a GOAP), then you can consider inheriting from the AStarPlanner, or Planner class, and defining your own "planning world" by inheriting from IPlanningWorld. The GetTraversalCost function is particularly important to note, because this defines the heuristic for the planner.

Additionally, you can define your own planning success conditions by inheriting from the SuccessCondition class. The SuccessCondition determines when the plan is complete. For example, a path plan is complete when the current node equals the goal node, which you can see in the ReachedGoalNode_SuccessCondition class. However, you may want a more complicated success condition, such as only ending the plan once it finds a node that is near the goal node, or that is near the goal node, and close enough to another friendly agent.

Finally, if you want to define your own navigation targets, take a look at INavTarget. Right now SimplePath supports navigation to a target object, or position, but you can define your own type of target by inheriting from INavTarget.

