# Game creation

In this document a step-by-step guide is given on how to add a new game to Ruler of the Plane. This will be done in a tutorial where we will build a simple game as an example. This will highlight the necessary steps needed to take to create a 2D game in Unity and incorporate the new mini-game into the overall game Ruler of the Plane. We will go through the steps on how to create the game scenes in Unity, show some code examples of how to handle game logic and discuss a way to implement levels for the game.

Other than this tutorial, you can study the implementation of other mini-games and read the documentation on Unity and the game design of Ruler of the Plane. We encourage you to reuse parts of the Unity scenes for other games as well as the game art. This tutorial is in no way the definite method for creating a game in Unity, but can be used as example.

## Introduction

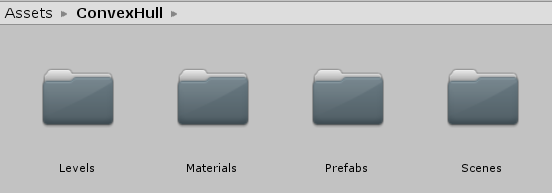
In this tutorial we will show how to create a custom game with a simple example based on convex hulls. This new game should show a set of points and the task for the player is to draw the convex hull around these points. Quite trivial, but it serves the purpose of the tutorial.

The game will re-use many parts of previous games, such as game art, prefabs and code snippets. This is encouraged in general. The codebase was separated into a game code and a decoupled geometric library for easier re-use, to accommodate the re-use of geometry algorithms and structures. The library already contains a Graham scan implementation, which we will use to calculate the convex hull.

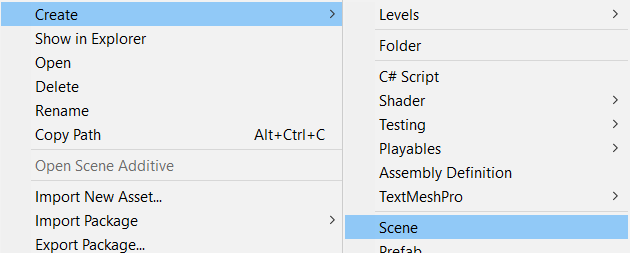
The creation of the game is separated into several distinct steps. First, the different game scenes need to be created in Unity. Likely you will want to create a menu scene as well as a main level scene. Next, some C# scripts need to be written to implement the game logic. After the base game is working, we will discuss how to implement level creation and importing.

## Initial setup

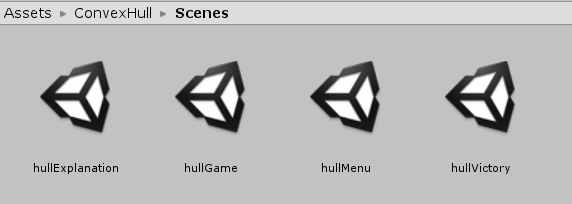
1. Create new game folder in ‘Assets’ called “ConvexHull”. Inside ‘Assets/ConvexHull’ create a new folder called “Scenes”. You can also already create folders called “Materials”, “Prefabs”, and “Levels”, since we will be needing those shortly.



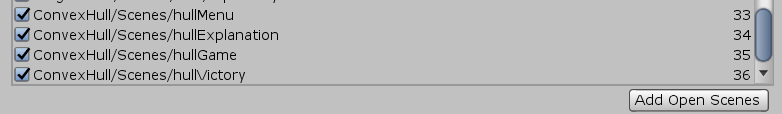
1. Go inside the ‘Assets/ConvexHull/Scenes” folder, right click and select ‘Create/Scene’. This should create a new scene object. For our convex hull game we will create a menu scene, an explanation scene, a game scene, and a victory scene. So go ahead and create four Unity scenes and name them accordingly. I used ‘hullMenu’, ‘hullExplanation’, ‘hullgame’, and ‘hullVictory’.



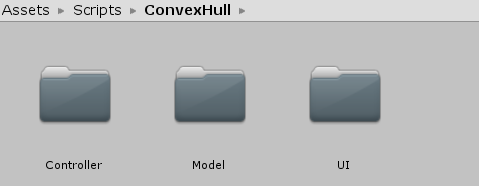
After creation your scenes folder should have four scenes show in the picture below. The idea is that player will first be shown the menu scene, after which it can either go to the explanation scene or start the actual game scene. If the game is completed, a small victory scene will be shown.



1. Add the four scenes to the Unity Build Settings. This is done by going to ‘File/Build Settings…’ in the top menu. Here you can add an open scene to the build settings, so open every scene by double clicking on the scene object and then press ‘Add Open Scenes’. Make sure all of the four scenes are shown at the end of the build order.



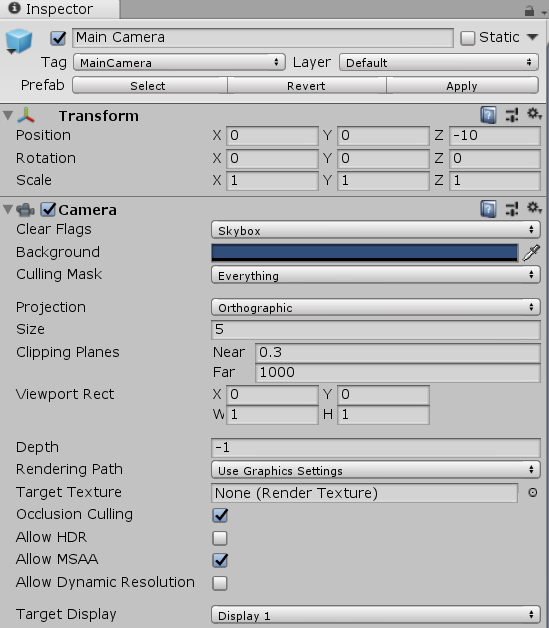
1. Create a new folder in ‘Assets/Scripts’ for this game, also called ‘ConvexHull’. Create subfolders ‘Controller’, ‘Model’, ‘UI’. Here we will add the associated C# scripts. For now we will just create empty scripts that inherit from MonoBehaviour. This is done by default if you create a C# script inside Unity by right clicking and pressing ‘Create/C# Script’. We will populate the scripts later.
   1. Create a controller script called ‘HullController.cs” in the subfolder ‘Controller’.
   2. Create a level scripts called ‘HullLevel.cs” in the subfolder ‘Model’.
   3. Create two scripts called ‘HullPoint.cs” and ‘HullSegment.cs’ in the subfolder ‘UI’.



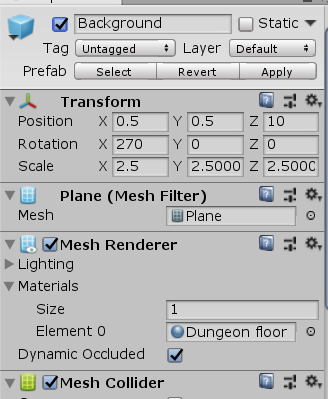
## Creating game scenes

### Menu Scene

1. We will first create a simple menu scene. So go ahead and open the scene ‘hullMenu’ in ‘Assets/ConvexHull/Scenes’ we created earlier. This is the first scene shown to the player when starting the game, and should have buttons for starting the game, showing an explanation and going back to the menu.
2. The initial scene should only have a camera object. The default settings (orthographic camera) are mostly fine except for the following:
   1. You should uncheck ‘Allow HDR’.
   2. Give the z-position some negative value (I used -10). This will ensure that most objects will automatically appear in front of the camera.



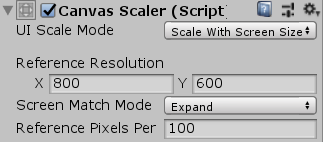
1. Add a background to the scene by right clicking inside the scene hierarchy view and selecting ‘3D object/Plane’. This should create a new object called ‘Plane’ in the scene. Now do the following:
   1. You can rename the plane to ‘Background’.
   2. The plane will not show up on camera at first, so add a rotation of 270 degrees around the x-axis.
   3. Add a positive value (> 0) to the z-position. This will ensure the background is behind most objects.
   4. Add a material to the ‘Mesh Renderer’ instead of the default. You can re-use a materials from one of the other games. Make sure to duplicate the original material (CTRL + D) and place it in the associated ‘ConvexHull/Materials’ folder. I chose the dungeon floor material from the Art Gallery game.



So far, your scene should look as follows:



1. Now we will create the menu items. Add a ‘Canvas’ object by right clicking in the scene hierarchy view and selecting ‘UI/Canvas’. This will create a new object called ‘Canvas’ as well as an object called ‘EventSystem’. Do not worry about the Event System for now. The Canvas will be the UI parent object which will hold all our UI items like buttons and the title.
   1. Inside the canvas object, set the UI Scale Mode of the Canvas Scaler to ‘Scale with Screen Size’. This will allow the UI to scale easily with different resolutions.
   2. In the Canvas Scaler, set Screen Match Mode to Expand. This will stop the UI elements from growing once the reference resolution is reached (the screen will simple expand).



So far, your scene should look as follows:



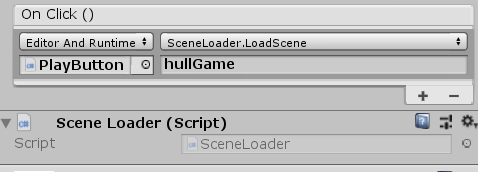
1. Add the menu buttons to the scene. To add a button, you can find the button prefab underneath ‘Assets/Menu/Prefabs’. Drag this prefab on top of the canvas object to instantiate it. We will create three buttons this way. For the button you should do the following:
   1. Change the text child object of the buttons to something like ‘Play game’, ‘Explanation’, and ‘Back to menu’.
   2. (Optional) Rename the button objects based on their text, so it’s easier to figure out which button is which. I chose ‘GameButton’, ‘ExplanationButton’, and ‘MenuButton’.
   3. Move the buttons to separate positions. Set the rectangle anchor point of each button correctly (top-middle-bottom, left-center-right) by pressing the image shown below.



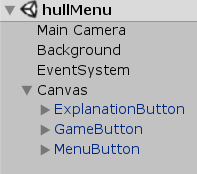
This defines the “reference” or anchor point that the position vector is based on. The ‘Play’ and ‘Explanation’ buttons should have middle-center preset and the ‘Back’ button bottom-center. Do this such that the UI will scale correctly with the screen. Next, change the position vector of the button to something appealing.

* 1. Load a new scene on a button click. The button prefabs will have a SceneLoader.cs script attached that can be used for loading new scenes. All you need to do is write the new scene name in the ‘On Click ()’ function of each button. For the three buttons these will be ‘hullgame’, ‘hullExplanation’, and ‘mainMenu’.

See picture below for the play button.

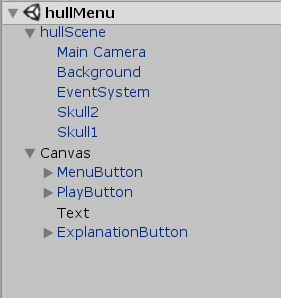


So far, your scene should look as follows:



1. Add a title to the scene. Right click on the canvas object and select ‘UI/Text’. Then do the following:
   1. Take a font from the ‘Assets/Menu’ folder, for example ‘eaglelake’.
   2. Set the anchor point to middle-top and change the position vector to your liking.
   3. Change the font size to something large (I used 72).
   4. Change the text to ‘Convex Hull’.
   5. Set the vertical and horizontal overflow fields to ‘Overflow’.
2. (Optional) Add some nice sprites to the scene to make it less boring. I used the skull sprite that is inside the ‘Assets/General/Textures/pois’ folder. Simply drag the sprite asset to the hierarchy view and it will create a sprite object. Then you can change the position, rotation, and scale to your liking.

The final menu screen should look something like the picture below. Try running the menu scene and pressing some of the buttons. If everything is set up correctly and the next scenes are present in the Build Settings, the new scene should be loaded (though the game and explanation scene will be empty).



1. For the future, you can create a prefab for the camera, background plane, the EventSystem, and potentially the two skull sprites, for re-use in the other scenes saving you a lot of effort. In the picture above I did this by creating the ‘HullScene’ object, which I stored as a prefab inside the ‘Assets/ConvexHull/Prefabs’ folder. Do this as follows:.
   1. Create an empty object in the scene by pressing right click and select ‘Create Empty’.
   2. (Optional) Give the object a fitting name.
   3. Add the camera, background plane, the EventSystem object, and the two skull sprites as children of the new object. Do this by dragging them on top of the new empty object.
   4. Create a prefab by dragging the object into the Assets/ConvexHull/Prefabs’ folder.

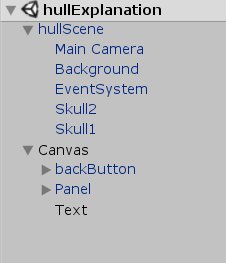


### Explanation Scene

We will now create the explanation scene. Here we will create a scene that gives a short explanation on the game and on convex hulls. Since the steps are largely the same as for the menu, we will go over them somewhat quicker.

1. Open the scene ‘hullExplanation’.
2. Delete the default camera object and instead drag the prefab object we made in the previous menu scene. This should contain a camera, background, event system and potentially some sprites. All that is left to do is to create the UI.
3. Add a canvas object, similar to how this was done for the menu. Namely, right clicking in the scene hierarchy view and selecting ‘UI/Canvas’. Set the scale mode to scale with screen and the screen match mode to expand.
4. Add a back button to the scene. Similar to how this was done for the menu. Namely, drag the button prefab at ‘Assets/Menu/Prefabs’ to the scene.
   1. Change the text to something like ‘Go back’.
   2. Set the anchor point to bottom-center and change the positions
   3. Change the scene name in the On Click () function to ‘hullMenu’.
5. Add a title to the scene, similar to how this was done for the menu. Namely, add a text object to the canvas object by right clicking and selecting ‘UI/Text’. Change the font, size and positioning accordingly and pick a nice title (I chose “Explanation”). One could also create a prefab from the title in the menu scene to make this step easier.
6. Add a text panel to the scene. Pick the panel prefab from the ‘Assets/Menu/Prefab and drag it on top of the canvas object. You don’t need to change much of the default settings. Write a nice explanation of the game and convex hulls in the associated child text object.

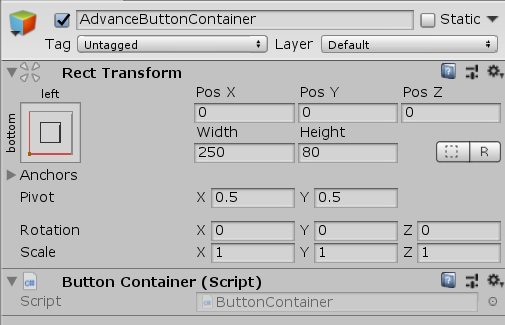
The final explanation scene should look something like the picture below. Try running the scene and pressing the back button.

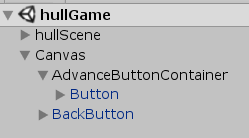
 

### Game Scene

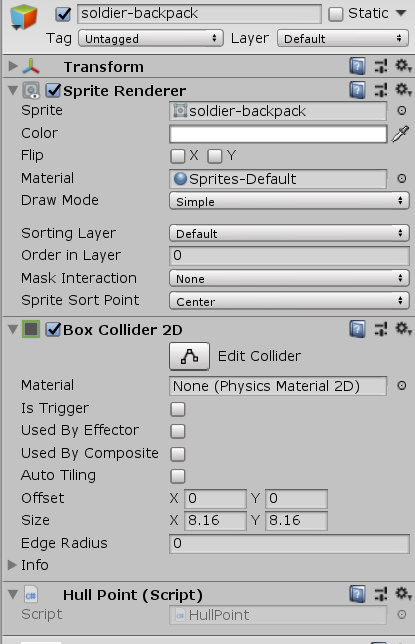
We will now create the main game scene. This is the scene “where the magic happens”, meaning the scene where the actual game takes place. Again, the main steps are similar to that for the menu and explanation scenes.

1. Open the scene ‘hullGame’.
2. Delete the default camera object and instead drag the ‘hullScene’ prefab we made in the previous menu scene. This should create the default background with potentially some sprites.
3. Add a canvas object, similar to how this was done previously.
4. Add two buttons to the scene, an advance button and a back button. Use the button prefab just as before.
   1. Set the anchor for the advance button to bottom-left and change the text to ‘Advance’. For now we will set the on click function to load the victory scene: ‘hullVictory’. This will be changed later.
   2. Set the anchor for the back button to bottom-right and change the text to ‘Go back’. Make it load the ‘hullMenu’ scene on a click.
5. Add an empty object (right click in the scene hierarchy and select ‘Create empty’) inside the canvas object, and do the following:
   1. Call the object ‘AdvanceButtonContainer’.
   2. Add the advance button as a child and make sure the name of the object is simply ‘Button’ (this will be important for the script we add in the next step).
   3. Add the ‘ButtonContainer’ script from ‘Assets/Scripts/General/Menu’ to the container. This will disable the button initially and allow us to enable in when the solution is found.



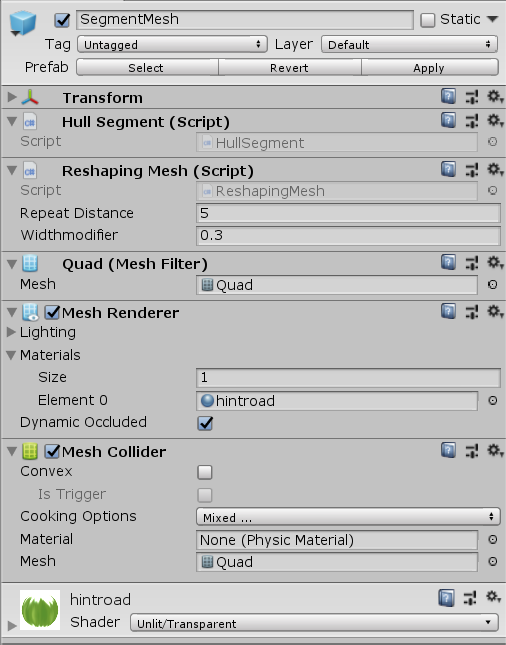


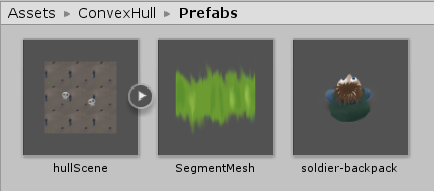
1. Create a prefab for a point.
   1. Take a sprite you like from ‘Assets/General/Textures’ and drag it into the scene hierarchy view. This will create an object with the sprite. (I chose the soldier with a backpack, just for fun).
   2. Add the ‘HullPoint.cs’ script from the ‘Assets/Scripts/ConvexHull/UI’ we created earlier to the object. Do this by finding the script and dragging it into the Unity Inspector for the sprite object.
   3. Add a Box Collider 2D component to the object, by pressing ‘Add Component’ at the bottom and finding it in the list. Unselect the checkbox ‘Used by composite’.
   4. Create a prefab by dragging the object to the ‘Assets/ConvexHull/Prefabs’ folder.



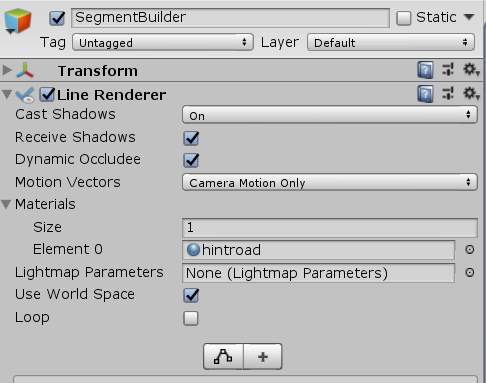


1. Create a prefab for a segment.
   1. Create a mesh object by right clicking inside the scene hierarchy view and selecting ‘3D Object/Quad’.
   2. (Optional) Rename it to ‘SegmentMesh’
   3. Add a road material to the ‘Mesh Renderer’ materials field. Do this by dragging a material object to the materials field. You can find road materials at ‘Assets/General/Textures/Border/Materials’. I chose the ‘hintRoad’.
   4. Add the ‘ReshapingMesh.cs’ script from ‘Assets/Scripts/General/Model’ to the object.
   5. Add the ‘HullSegment.cs’ script from ‘Assets/Scripts/ConvexHull/UI’ to the object.
   6. Create the prefab by dragging the object to the prefabs folder.



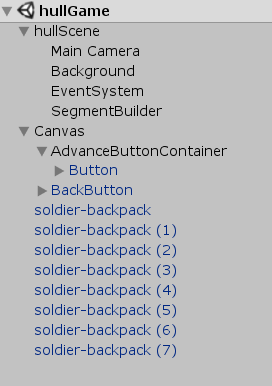


1. Add the game controller script (‘HullController.cs’ at ‘Assets/Scripts/ConvexHull/Controller’) to a game object. The exact object does not really matter, so choose something that makes sense. I chose the root ‘hullScene’ object.
2. Add an empty object (right click in the scene hierarchy and select ‘Create empty’) to the scene and add a Line Renderer component.
   1. (Optional) Rename the object ‘SegmentBuilder’.
   2. Add the road material to the ‘Line Renderer’ materials field. Use the same road material you selected for the hull segment prefab.



1. Add some soldier objects to the scene to create an initial level. You can remove the original prefab objects for the soldier and road. Than populate the scene with the soldier prefab. Later we will talk about how to create level objects, but for now we will just test with a single hardcoded level.

The final game scene should look something like the picture below. Try running the game and see if the advance and back buttons work.

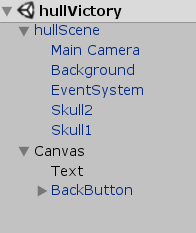
 

### Victory Scene

Finally, we will create a very simple victory scene that shows a nice message after completing the game. By now, you should know how to do this yourself, but here are the quick steps.

1. Open the scene ‘hullVictory’.
2. Delete the default camera object and instead drag the prefab object we made in the previous menu scene.
3. Add a canvas object.
4. Add a back button to the scene and link it to the ‘hullMenu’ scene.
5. Add a title to the scene.

The final victory scene should look something like the picture below. Feel free to add some more sprites or decoration to make the scene more appealing, to really show some appreciation to the player.

## Writing game scripts

We have now created the main scenes of the game, and linked them together using some buttons. Now, it is time to create the actual game. For this we will need to write the game logic using a few C# scripts. Since this tutorial is not really focused on C#, I will simply show how I implemented the different scripts. Feel free to change parts or implement your own version, since there are definitely ways to improve my implementation.

1. We will implement the main game logic with a game controller that handles the game loop and two UI elements that handle user input (namely clicks on points and clicks on segments):
   1. The HullController contains the main game loop. Furthermore, it is responsible for creating and deleting the line segments between the points, as well as checking if the correct solution is found. Initially, it will disable the advance and re-enable it when the correct hull is built by the user. At first, it will retrieve all hull points present in the scene to calculate a solution convex hull. Every frame it will check for key presses to see whether it needs to create a segment. Whenever a segment is added or removed, it will check the player’s segments with the solution hull.
   2. The HullPoint will accept user input like mouse enter, exit, or click, and change some variables in the game controller related to segment creation.
   3. The HullSegment will only listen for user clicks to handle user removal of segments.

See the scripts below for my implementation of the game logic. You can directly take this code and put it in your own files or use it as an example to create your own implementation.

#### HullController.cs

namespace ConvexHull

{

using General.Menu;

using General.Model;

using System.Collections.Generic;

using System.Linq;

using UnityEngine;

using Util.Geometry.Polygon;

using Util.Algorithms.Polygon;

using Util.Geometry;

public class HullController : MonoBehaviour

{

public LineRenderer m\_line;

[SerializeField]

private GameObject m\_roadMeshPrefab;

[SerializeField]

private ButtonContainer m\_advanceButton;

internal HullPoint m\_firstPoint;

internal HullPoint m\_secondPoint;

internal bool m\_locked;

private List<HullPoint> m\_points;

private HashSet<LineSegment> m\_segments;

private Polygon2D m\_solutionHull;

void Start()

{

// get unity objects

m\_points = FindObjectsOfType<HullPoint>().ToList();

m\_segments = new HashSet<LineSegment>();

// compute convex hull

m\_solutionHull = ConvexHull.ComputeConvexHull(m\_points.Select(v => v.Pos));

// disable advance button

m\_advanceButton.Disable();

}

void Update()

{

if (m\_locked && !Input.GetMouseButton(0))

{

// create road

AddSegment(m\_firstPoint, m\_secondPoint);

}

else if (Input.GetMouseButton(0))

{

// update road endpoint

var pos = Camera.main.ScreenToWorldPoint(Input.mousePosition + 10 \* Vector3.forward);

m\_line.SetPosition(1, pos);

}

// clear road creation variables

if ((m\_locked && !Input.GetMouseButton(0)) || Input.GetMouseButtonUp(0))

{

m\_locked = false;

m\_firstPoint = null;

m\_secondPoint = null;

m\_line.enabled = false;

}

}

public void AddSegment(HullPoint a\_point1, HullPoint a\_point2)

{

var segment = new LineSegment(a\_point1.Pos, a\_point2.Pos);

// dont add double segments

if (m\_segments.Contains(segment) || m\_segments.Contains(new LineSegment(a\_point2.Pos, a\_point1.Pos)))

// also check reverse

return;

m\_segments.Add(segment);

// instantiate new road mesh

var roadmesh = Instantiate(m\_roadMeshPrefab, Vector3.forward, Quaternion.identity) as GameObject;

roadmesh.transform.parent = this.transform;

roadmesh.GetComponent<HullSegment>().Segment = segment;

var roadmeshScript = roadmesh.GetComponent<ReshapingMesh>();

roadmeshScript.CreateNewMesh(a\_point1.transform.position, a\_point2.transform.position);

CheckSolution();

}

public void RemoveSegment(HullSegment a\_segment)

{

m\_segments.Remove(a\_segment.Segment);

CheckSolution();

}

public void CheckSolution()

{

if (CheckHull())

{

m\_advanceButton.Enable();

}

else

{

m\_advanceButton.Disable();

}

}

private bool CheckHull()

{

// quick return counts not equal

if (m\_solutionHull.Segments.Count != m\_segments.Count)

return false;

return m\_solutionHull.Segments.All(seg => m\_segments.Contains(seg) ||

m\_segments.Contains(new LineSegment(seg.Point2, seg.Point1))); // also check reverse

}

}

}

#### HullPoint.cs

namespace ConvexHull

{

using UnityEngine;

public class HullPoint : MonoBehaviour

{

public Vector2 Pos { get; private set; }

private HullController m\_controller;

void Awake()

{

Pos = new Vector2(transform.position.x, transform.position.y);

m\_controller = FindObjectOfType<HullController>();

}

void OnMouseDown()

{

m\_controller.m\_line.enabled = true;

m\_controller.m\_firstPoint = this;

m\_controller.m\_line.SetPosition(0, Pos);

}

void OnMouseEnter()

{

if (m\_controller.m\_firstPoint == null) return;

m\_controller.m\_locked = true;

m\_controller.m\_secondPoint = this;

m\_controller.m\_line.SetPosition(1, Pos);

}

void OnMouseExit()

{

if (this != m\_controller.m\_secondPoint) return;

m\_controller.m\_locked = false;

m\_controller.m\_secondPoint = null;

var pos = Camera.main.ScreenToWorldPoint(Input.mousePosition + 10 \* Vector3.forward);

m\_controller.m\_line.SetPosition(1, pos);

}

}

}

#### HullSegment.cs

namespace ConvexHull

{

using UnityEngine;

using Util.Geometry;

public class HullSegment : MonoBehaviour

{

public LineSegment Segment { get; set; }

private HullController m\_gameController;

void Awake()

{

m\_gameController = FindObjectOfType<HullController>();

}

void OnMouseUpAsButton()

{

// destroy the road object

m\_gameController.RemoveSegment(this);

Destroy(gameObject);

}

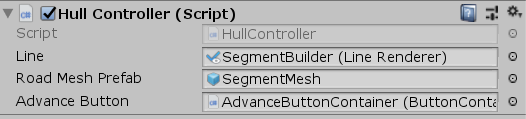
}

}

## Configuring game scene

If you have used the code given before, you will need to configure the serializable fields of the game controller with the correct unity objects. See the following step.

1. Go back to the Unity ‘hullGame’ scene and select the object you have attached the ‘HullController’ script to. Now attach game objects to the following fields:
   1. Attach the road mesh prefab you have created earlier.
   2. Attach the button container that has the advance button.
   3. Attach the object containing the line renderer.



After this step the game should be all set up to play the first (and only) level. Try running the game scene and see if you can create segments between the hull points. Do this by pressing on one point and dragging the mouse to another point. This should create a segment between the two points. Doing this, you can draw the convex hull around your points. If everything was done correctly, the advance button should be enabled once the convex hull is drawn. After pressing the advance button, you should be greeted by the victory scene

## Adding levels

We have now created a working game inside the Ruler of the Plane, but it only consists of a single hardcoded level. It would thus be nice to add some more levels to the game. One way would be to create a new game scene for each level and linking the buttons accordingly, but this will result in many duplicate scenes. So we will want to create separate level objects and load these into the main game scene. This will also make importing levels easier.

### Level script

The data for each level should stored inside a scriptable object class, which acts as a data container class in Unity. For a new type of level, one needs to create a custom class that inherits from ‘ScriptableObject’. Inside the class all data is placed in serializable fields (public fields of “simple” data types are by default serializable) that are necessary to describe the level. These scriptable objects can then be used inside the game controller to initialize a new level.

1. Implement the C# script ‘ConvexLevel’ you created earlier inside ‘Assets/ConvexHull/Model’.
   1. The class should implement from the scriptable object interface (a Unity data container class).
   2. Add a field for the point set, simply as a list of vectors. You can optionally add a header for extra clarity in the Unity Inspector.
   3. Add the ability for manual creation in Unity. Add the class attribute ‘CreateAssetMenu’ above the class. This will allow the scriptable object to be listed under the ‘Assets/Create’ menu. You should give parameters for the default file name of the newly created object and give a name inside the menu for the user to select.

See the code below for my implementation of the level object.

#### HullLevel

namespace ConvexHull

{

using System.Collections.Generic;

using UnityEngine;

/// <summary>

/// Data container for convex hull level, containing point set.

/// </summary>

[CreateAssetMenu(fileName = "hullLevelNew", menuName = "Levels/Convex Hull Level")]

public class HullLevel : ScriptableObject

{

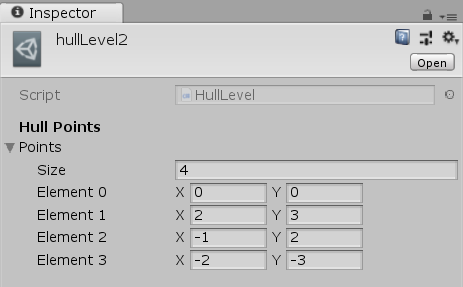
[Header("Hull Points")]

public List<Vector2> Points = new List<Vector2>();

}

}

You can test the level object by manually creating a hull level. Do this by right clicking inside a folder and selecting ‘Create/Level/Convex Hull Level’ (or whatever you specified in ‘CreateAssetMenu’). A new HullLevel object should appear where you can manually input points.



### Adjust game controller

To accommodate the game for the use of levels, we need to make some changes to the game controller. Instead of extracting the hull points from the scene, it needs to instantiate them from a level object. This means the game controller should have a list of levels, and methods to initialize a level and advance to the next one. It now also needs to deal with cleanup of instantiated objects when advancing to the next level, to avoid objects persisting between levels.

1. Adjust the controller for the use of levels. You can use the code shown below. In bold are all lines that changed:

#### HullController

namespace ConvexHull

{

using General.Menu;

using General.Model;

using System.Collections.Generic;

using System.Linq;

using UnityEngine;

using Util.Geometry.Polygon;

using Util.Algorithms.Polygon;

using Util.Geometry;

**using General.Controller;**

**using UnityEngine.SceneManagement;**

public class HullController : MonoBehaviour, **IController**

{

public LineRenderer m\_line;

[SerializeField]

private GameObject m\_roadMeshPrefab;

**[SerializeField]**

**private GameObject m\_pointPrefab;**

[SerializeField]

private ButtonContainer m\_advanceButton;

**[SerializeField]**

**private List<HullLevel> m\_levels;**

**[SerializeField]**

**private string m\_victoryScene;**

internal HullPoint m\_firstPoint;

internal HullPoint m\_secondPoint;

internal bool m\_locked;

private List<HullPoint> m\_points;

private HashSet<LineSegment> m\_segments;

private Polygon2D m\_solutionHull;

**private List<GameObject> instantObjects;**

**protected int m\_levelCounter = 0;**

void Start()

{

// get unity objects

**m\_points = new List<HullPoint>();**

m\_segments = new HashSet<LineSegment>();

**instantObjects = new List<GameObject>();**

**InitLevel();**

}

void Update()

{

if (m\_locked && !Input.GetMouseButton(0))

{

// create road

AddSegment(m\_firstPoint, m\_secondPoint);

}

else if (Input.GetMouseButton(0))

{

// update road endpoint

var pos = Camera.main.ScreenToWorldPoint(Input.mousePosition + 10 \* Vector3.forward);

m\_line.SetPosition(1, pos);

}

// clear road creation variables

if ((m\_locked && !Input.GetMouseButton(0)) || Input.GetMouseButtonUp(0))

{

m\_locked = false;

m\_firstPoint = null;

m\_secondPoint = null;

m\_line.enabled = false;

}

}

**public void InitLevel()**

**{**

**if (m\_levelCounter >= m\_levels.Count)**

**{**

**SceneManager.LoadScene(m\_victoryScene);**

**return;**

**}**

**// clear old level**

**Clear();**

**// initialize settlements**

**foreach (var point in m\_levels[m\_levelCounter].Points)**

**{**

**var obj = Instantiate(m\_pointPrefab, point, Quaternion.identity) as GameObject;**

**obj.transform.parent = this.transform;**

**instantObjects.Add(obj);**

**}**

**//Make vertex list**

**m\_points = FindObjectsOfType<HullPoint>().ToList();**

**// compute convex hull**

**m\_solutionHull = ConvexHull.ComputeConvexHull(m\_points.Select(v => v.Pos));**

**m\_advanceButton.Disable();**

**}**

**public void AdvanceLevel()**

**{**

**// increase level index**

**m\_levelCounter++;**

**InitLevel();**

**}**

public void AddSegment(HullPoint a\_point1, HullPoint a\_point2)

{

var segment = new LineSegment(a\_point1.Pos, a\_point2.Pos);

// dont add double segments

if (m\_segments.Contains(segment) || m\_segments.Contains(new LineSegment(a\_point2.Pos, a\_point1.Pos)))

// also check reverse

return;

m\_segments.Add(segment);

// instantiate new road mesh

var roadmesh = Instantiate(m\_roadMeshPrefab, Vector3.forward, Quaternion.identity) as GameObject;

roadmesh.transform.parent = this.transform;

**instantObjects.Add(roadmesh);**

roadmesh.GetComponent<HullSegment>().Segment = segment;

var roadmeshScript = roadmesh.GetComponent<ReshapingMesh>();

roadmeshScript.CreateNewMesh(a\_point1.transform.position, a\_point2.transform.position);

CheckSolution();

}

public void RemoveSegment(HullSegment a\_segment)

{

m\_segments.Remove(a\_segment.Segment);

CheckSolution();

}

public void CheckSolution()

{

if (CheckHull())

{

m\_advanceButton.Enable();

}

else

{

m\_advanceButton.Disable();

}

}

private bool CheckHull()

{

// quick return counts not equal

if (m\_solutionHull.Segments.Count != m\_segments.Count)

return false;

return m\_solutionHull.Segments.All(seg => m\_segments.Contains(seg) ||

m\_segments.Contains(new LineSegment(seg.Point2, seg.Point1))); // also check reverse

}

**/// <summary>**

**/// Clears hull and relevant game objects**

**/// </summary>**

**private void Clear()**

**{**

**m\_solutionHull = null;**

**m\_points.Clear();**

**m\_segments.Clear();**

**// destroy game objects created in level**

**foreach (var obj in instantObjects)**

**{**

**// destroy immediate**

**// since controller will search for existing objects afterwards**

**DestroyImmediate(obj);**

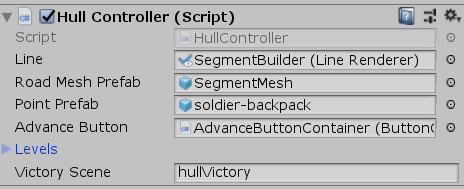
**}**

**}**

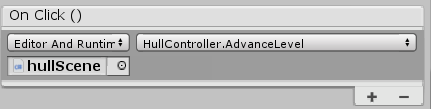
}

}

1. Correctly link the newly created game controller fields. After changing the controller, we have added the serializable fields m\_pointPrefab, m\_levels, and m\_victoryScene. The levels we will create later, but for now you can link the point prefab in ‘Assets/ConvexHull/Prefabs’ and set the victory scene string to ‘hullScene’.



1. Change the advance button to call the hullController AdvanceLevel() instead of going to the victory scene. Do this by dragging the game object containing the game controller script into the On Click () box and selecting the Advance Level function to call.



### Importing (Optional)

If possible, it is encouraged to allow the user to import IPE level files instead of manual inserting level data. This makes creating large levels easier than the manual method of inserting coordinates.

We need to decide on how we want our convex hull to be created inside IPE. The IPE file should contain the point set for the convex hull level. The most natural object to represent a point is a marker point (icon: ). Thus, I decided to interprete a ‘disk’ marker as a point for the level.

1. Add a call inside the LoadLevelEditor script to create a hull level object. Open the LoadLevelEditor.cs script located at ‘Assets/Scripts/Editor’. Move to the method ‘OnImportAsset(AssetImportContext ctx)’, which is called upon an IPE file import. Adjust the “switch” statement between level creation based on name to call a new function called ‘LoadHullLevel’, similar to how this is done for the other levels. See the code snippet below for more details.

public override void OnImportAsset(AssetImportContext ctx)

{

var path = ctx.assetPath;

var name = Path.GetFileNameWithoutExtension(path);

var fileSelected = XElement.Load(path);

// switch between which level to generate based on file name

Object obj;

if (name.StartsWith("agLevel"))

{

obj = LoadArtGalleryLevel(fileSelected, name);

}

else if (name.StartsWith("ktLevel"))

{

obj = LoadKingsTaxesLevel(fileSelected, name);

}

else if (name.StartsWith("divLevel"))

{

obj = LoadDivideLevel(fileSelected, name);

}

**else if (name.StartsWith("hullLevel"))**

**{**

**obj = LoadHullLevel(fileSelected, name);**

**}**

else

{

// no file name match

EditorUtility.DisplayDialog("Error", "Level name not in an expected format", "OK");

ctx.SetMainObject(null);

return;

}

// add generated level as the main imported file

ctx.AddObjectToAsset(name, obj);

ctx.SetMainObject(obj);

}

1. Write the ‘LoadHullLevel’ that creates a level from the XML file (and possible the file name). The function should create a scriptable object for the HullLevel, and retrieve the point set from all disk markers present in the IPE file. See the code snippet below for an implementation. I added a normalization that scales the points to fill the screen.

private Object LoadHullLevel(XElement fileSelected, string name)

{

// create the output scriptable object

var asset = ScriptableObject.CreateInstance<HullLevel>();

// retrieve page data from .ipe file

var items = fileSelected.Descendants("page").First().Descendants("use");

// get marker data into respective vector list

asset.Points.AddRange(GetMarkers(items, "disk"));

// normalize coordinates

var rect = BoundingBoxComputer.FromPoints(asset.Points);

Normalize(rect, ktSIZE, ref asset.Points);

// give warning if no relevant data found

if (asset.Points.Count == 0)

{

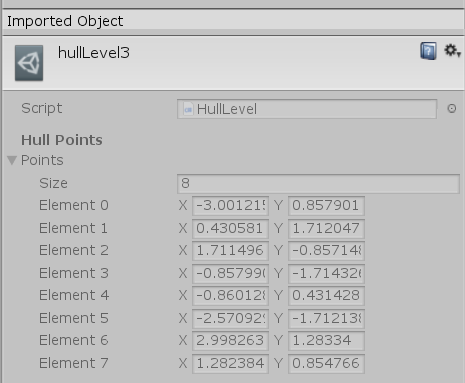
EditorUtility.DisplayDialog("Warning", "File does not contain any villages/castles (disks and/or squares).", "OK");

}

return asset;

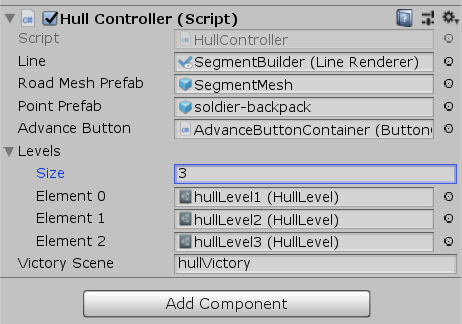
}

You should now be able to import an IPE file that represents a hull level. Create a new IPE file, place many disk marker point (press M to place) and save it with a name starting in ‘hullLevel’. If you import this file (either by dragging it into a folder in Unity or right clicking and selecting ‘Import New Asset…’) you should see a new imported object of HullLevel having been created.



### Create levels

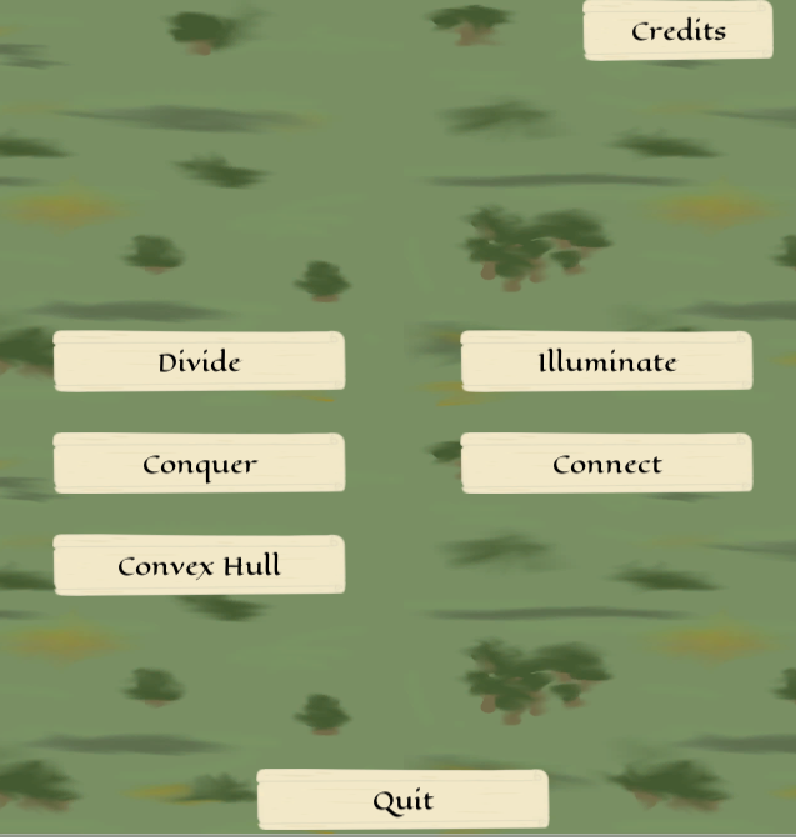
1. Create a new folder in ‘Assets/ConvexHull’ called ‘Levels’. Here we can store all the convex hull level objects.
2. Add some nice levels to the ‘Assets/ConvexHull/Levels, either manually through right clicking and selecting ‘Create/Levels/Convex Hull’ or by importing an IPE file with a collection of disk marker points.
3. Go to the main game scene ‘hullGame’ and select the object that holds the controller script. Here we should add the level objects to the level fields, see the picture below.



After this step, you should be able to run the game again, now with custom levels. The game controller should initialize each level in order according to the level objects, and advance to the next level whenever a solution was found. Check carefully if all instantiated objects were deleted after advancing to the next level.

## Main menu

1. Finally, we would like our game to be reachable from the main menu. So open the mainMenu scene located at ‘Assets/Menu/Scenes’ and add a menu button to the canvas that links to the game (‘hullMenu’). Adding a button can be done using the button prefab, just like before.



Congratulations, you have now added your own game with custom levels to the Ruler of the Plane Unity project!