CT3536 (Games Programming) Section 8

Unity Audio
Saving data with PlayerPrefs
Fun with Trigonometry
Some Game Dev Patterns & Techniques
Runtime Efficiency in Unity

Audio in Unity

- Unity provides 3D (spatialized) audio with configurable attenuation settings, as well as dynamic runtime effects such as reverb
- **AudioClip** - this is associated with a sound file (typically wav, ogg, or mp3)
- It isn't a MonoBehaviour/component, but rather it's a simple data/media object which needs an AudioSource to actually play it (in the same way that a JPG is not the same as a sprite)
- **AudioSource** – a component attached to a game object which can emit audio. An AudioSource has one specific AudioClip associated with it at a time (this might remain fixed or it might change as the game runs). Methods and properties:
  - Play(), Stop(), Pause() methods
  - isPlaying (Boolean)
  - clip (reference to AudioClip object it’s currently using)
  - loop (Boolean)
  - Volume (in the range 0.0 to 1.0)
- **AudioListener** – a component; there should be one (and only one) of these instantiated at a time. Typically attached to the object being controlled by the player (or to the camera)

- Example - my AudioObjectSimple class (see next slides)

```
// AudioObjectSimple only plays one sound at a time, either randomly from its list or by id number

public class AudioObjectSimple : MonoBehaviour {

    // inspector settings
    public float volume = 1f;
    public bool varyPitch = true;
    public List<AudioClip> sounds = new List<AudioClip>();
    //

    private AudioSource source;

    void Awake () {
        source = gameObject.AddComponent<AudioSource>();
        source.maxDistance = 7f;
        source.spatialBlend = 1f; // full 3d
        source.rolloffMode = AudioRolloffMode.Custom; // if we leave it at the default (logarithmic) the max distance isn't where volume stops, it's where it stops attenuating
        source.volume = volume;
    }
    public void PlayRandomSound(int lowestIdx=0) {
        if (sounds.Count==1)
            source.clip = sounds[0];
        else
            source.clip = sounds[Random.Range(lowestIdx, sounds.Count)];

        if (varyPitch)
            source.pitch = Random.Range(0.833f, 1.2f);

        source.Play();
    }

    public void PlaySoundByIndex(int idx) {
         if (idx<sounds.Count) {
            source.clip = sounds[idx];
            if (varyPitch)
                source.pitch = Random.Range(0.91f, 1.1f);
            source.Play();
         }
    }

} // end of class
```

Multiple sounds playing at once

```
public class MyScript : MonoBehaviour {

    private List<AudioSource> audioSources = new List<AudioSource>();


    private AudioSource GetAudioSource() {
        for (int i=0; i<audioSources.Count; i++) {
            if (!audioSources[i].isPlaying) {
                audioSources[i].loop = false;
                return audioSources[i];
            }
        }
        AudioSource asrc = gameObject.AddComponent<AudioSource>();
        audioSources.Add(asrc);
        asrc.playOnAwake = false;
        asrc.dopplerLevel = 0f; // valid range is 0-5
        asrc.loop = false;
        return asrc;
    }

}
```

AudioSource.PlayClipAtPoint() static method

- This is a handy way to play a single sound clip at an arbitrary location in the 3D world (not attached to any moving object)

- https://docs.unity3d.com/ScriptReference/AudioSource.PlayClipAtPoint.html

- AudioSource.PlayClipAtPoint(AudioClip clip, Vector3 position, float volume = 1.0f);

- You need to have an instantiated AudioClip, of course. The easiest way is via a public member in a script (i.e. assignable by dragging in the inspector) - or perhaps an array of them (as above in AudioObjectSimple)

Saving (persisting) data to disk

- You can use the standard C# file handling class ‘File’
- Binary or Text files
- You can save to:
  - Application.persistentDataPath + “/file.txt”
- Which maps to a directory your app can access, e.g. on Windows:
  - C:\Users\Sam\AppData\LocalLow\PsychicSoftware\DemonPit\file.txt

PlayerPrefs

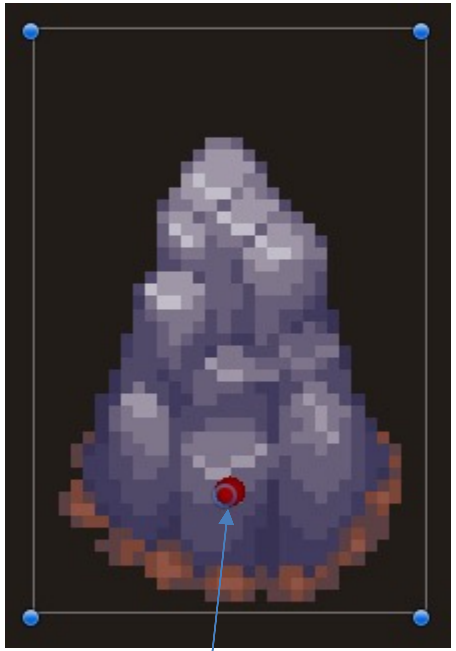
- Static methods of the PlayerPrefs class provide a very easy way to save and load small pieces of data, to persist between game sessions
- Stored in a device/OS-specific place, e.g.:
  - Windows: in the registry
  - Mac: in ~/Library/Preferences

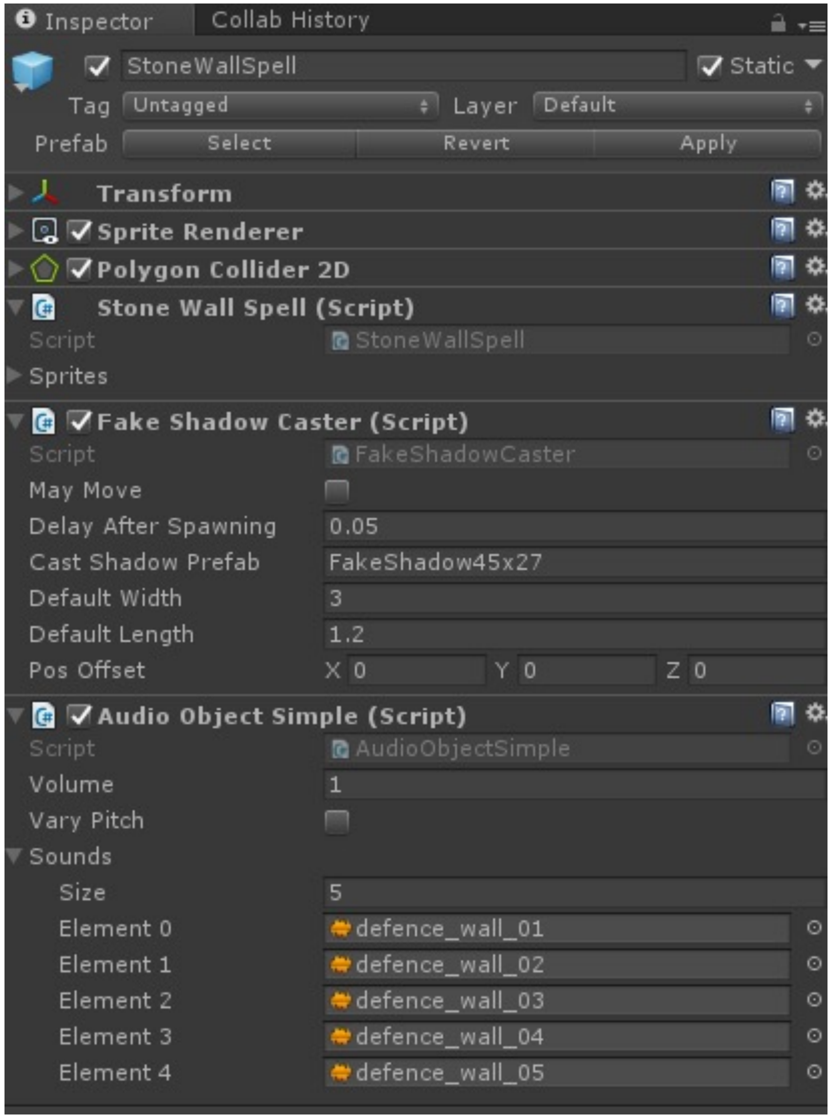
DeleteAll	Removes all keys and values from the preferences.
DeleteKey	Removes key and its corresponding value from the preferences.
GetFloat	Returns the value corresponding to key in the preferences if it exists.
GetInt	Returns the value corresponding to key in the preferences if it exists.
GetString	Returns the value corresponding to key in the preferences if it exists.
HasKey	Returns true if key exists in the preferences.
Save	Writes all modified preferences to disk (also occurs on normal quit)
SetFloat	Sets the value of the preference identified by key.
SetInt	Sets the value of the preference identified by key.
SetString	Sets the value of the preference identified by key.

Running code in the Editor (to assist the scene editing process, etc.)

(devtools are an important category of code that programmers write- to assist other members of the team rather than as part of the final game itself)

Editor Code





Inspector | Collab History

☑ StoneWallSpell ☑ Static

Tag: Untagged | Layer: Default

Prefab: Select | Revert | Apply

Transform

☑ Sprite Renderer

☑ Polygon Collider 2D

Stone Wall Spell (Script)

Script	StoneWallSpell
Sprites	

☑ Fake Shadow Caster (Script)

Script	FakeShadowCaster
May Move	☐
Delay After Spawning	0.05
Cast Shadow Prefab	FakeShadow45x27
Default Width	3
Default Length	1.2
Pos Offset	X 0 Y 0 Z 0

☑ Audio Object Simple (Script)

Script	AudioObjectSimple
Volume	1
Vary Pitch	☐
Sounds	
Size	5
Element 0	defence_wall_01
Element 1	defence_wall_02
Element 2	defence_wall_03
Element 3	defence_wall_04
Element 4	defence_wall_05

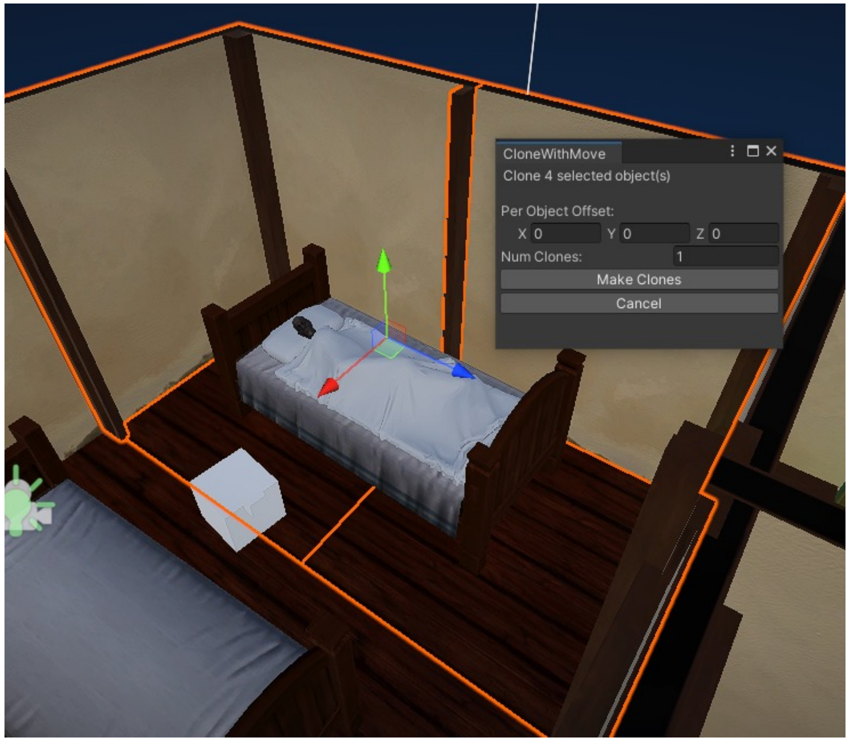
Unity can run code in the Editor (rather than the live game)
e.g. I have used OnDrawGizmosSelected() to draw a red sphere to help get posOffset set correctly, in my FakeShadowCaster script

```
void OnDrawGizmosSelected() {
    // debug draw in scene view only, when selected only
    Gizmos.color = Color.red;
    Gizmos.DrawSphere(transform.position+posOffset, 0.02f);
}
```

Editor code added to Unity menus

- You can write code to run in the editor whenever you require,
    - e.g. to add a collider to objects and size it correctly
    - e.g. to find all lights in the scene and change one of their settings
    - Or anything else which automates a manual task during scene editing
- If you want a method to be added to a Unity menu, its class script has to be stored in a folder named 'Editor'
- Now code can be run, at design-time, from a menu item, and can interact with the Scene and the Assets
- (See next slide)

```
using UnityEditor;

public class CloneWithMove : EditorWindow {
  Vector3 perObjectOffset;
  int numClones = 1;

  [MenuItem("Window/The Necromancer/Clone Selected Objects with Move %F3")]
  static void MoveAndClone() {
    CloneWithMove window = ScriptableObject.CreateInstance<CloneWithMove>();
    window.position = new Rect(Screen.width / 2, Screen.height / 2, 250, 160);
    window.Show();
  }

  private void OnGUI() {
    EditorGUILayout.LabelField("Clone "+ Selection.count +" object(s)", EditorStyles.wordWrappedLabel);
    GUILayout.Space(10);

    perObjectOffset = EditorGUILayout.Vector3Field("Per Object Offset:", perObjectOffset);

    numClones = EditorGUILayout.IntField("Num Clones:", numClones);

    if (GUILayout.Button("Make Clones")) {
      DoCloning();
      this.Close();
    }

    if (GUILayout.Button("Cancel"))
      this.Close();
    }

  void DoCloning() {
    foreach (GameObject go in Selection.gameObjects) {
      for (int num=1; num<=numClones; num++) {
        GameObject go2 = Instantiate(go);
        go2.transform.SetParent(go.transform.parent, false);
        go2.transform.localPosition =
            go.transform.localPosition + perObjectOffset * num;
      }
    }
  }
}
```



Clone 4 selected object(s)

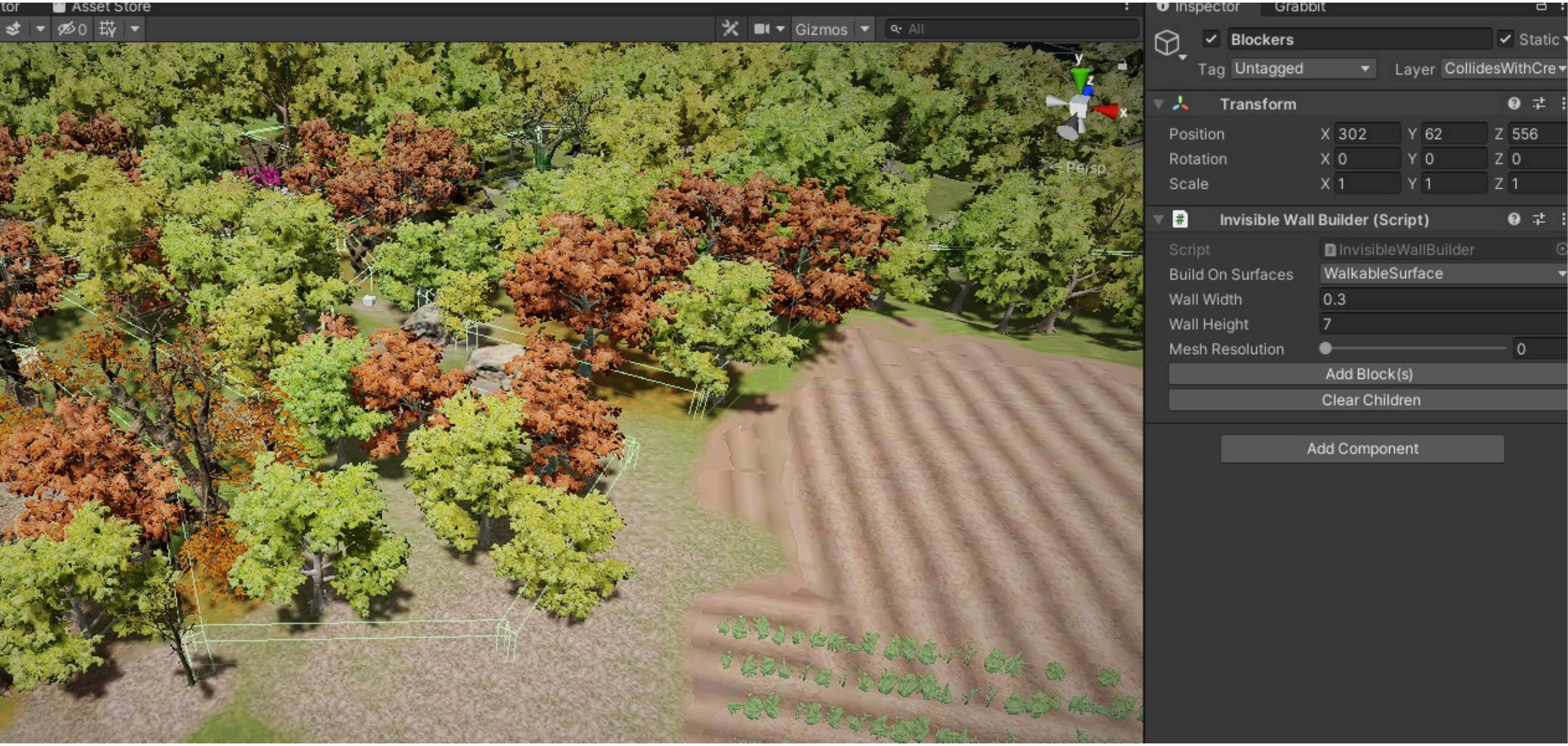
Per Object Offset:
X 0 Y 0 Z 0
Num Clones: 1

```
public class AddColliderToFitChildren : MonoBehaviour {

 [MenuItem("Window/The Necromancer/Add Collider To Fit Active Children")]
  static void FitToChildren() {
    foreach (GameObject rootGameObject in Selection.gameObjects) {
      bool hasBounds = false;
      Bounds bounds = new Bounds(Vector3.zero, Vector3.zero);

      Quaternion rot = rootGameObject.transform.rotation;
      rootGameObject.transform.rotation = Quaternion.identity;
      for (int i = 0; i < rootGameObject.transform.childCount; i++) {
        Transform t = rootGameObject.transform.GetChild(i);
        if (t.gameObject.activeInHierarchy) {
          Renderer childRenderer = t.GetComponent<Renderer>();
          if (childRenderer!=null) {
            if (hasBounds) {
              bounds.Encapsulate(childRenderer.bounds);
            }
            else {
              bounds = childRenderer.bounds;
              hasBounds = true;
            }
          }
        }
      }

      BoxCollider collider = rootGameObject.AddComponent<BoxCollider>();
      collider.center = bounds.center - rootGameObject.transform.position;
      collider.size = bounds.size;

      rootGameObject.transform.rotation = rot;
    }
  }
}
```

Remove rotation by setting it to Quaternion.identity before calculating minimum axis-aligned bounding boxes.

Revert after.

Invisible Wall Builder



Blockers
Static
Tag Untagged
Layer CollidesWithCre

Transform

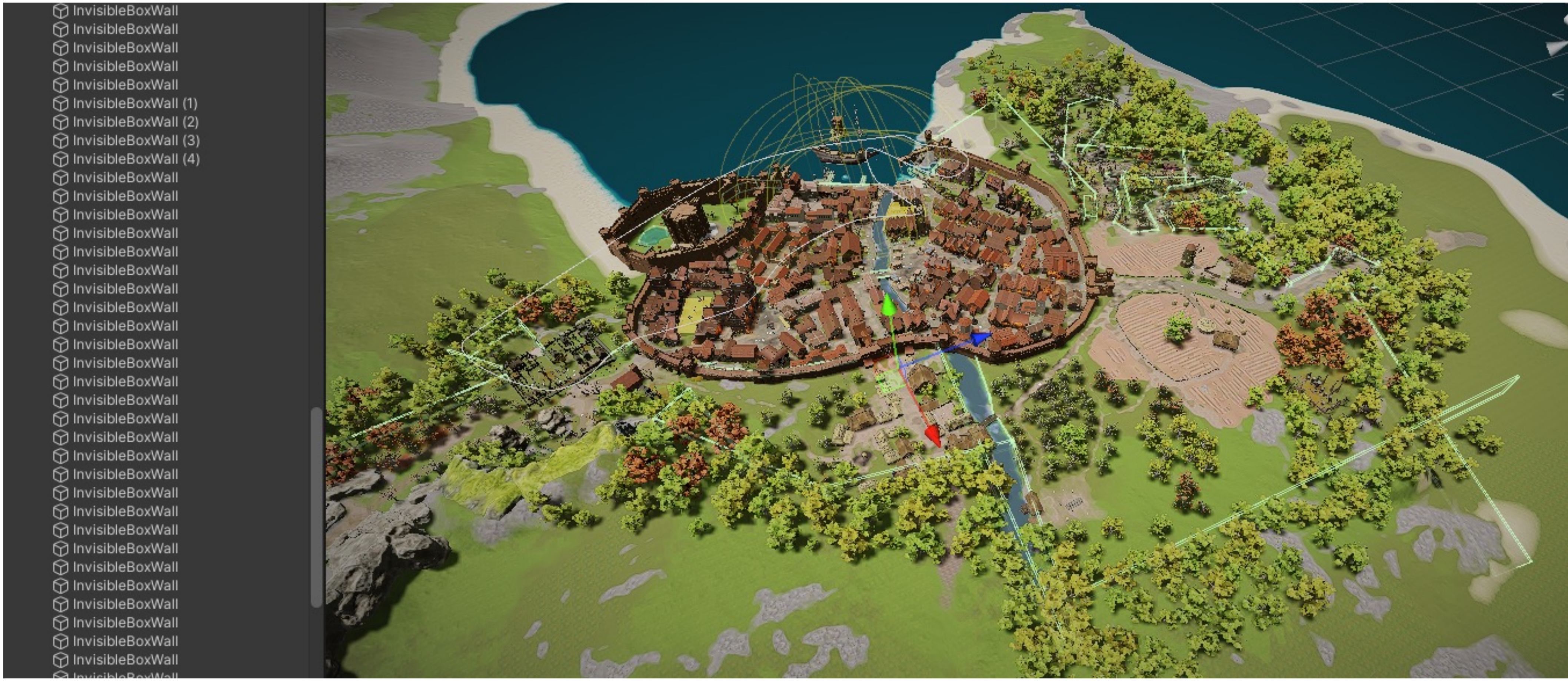
Position X 302 Y 62 Z 556
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1

Invisible Wall Builder (Script)

Script InvisibleWallBuilder
Build On Surfaces WalkableSurface
Wall Width 0.3
Wall Height 7
Mesh Resolution 0

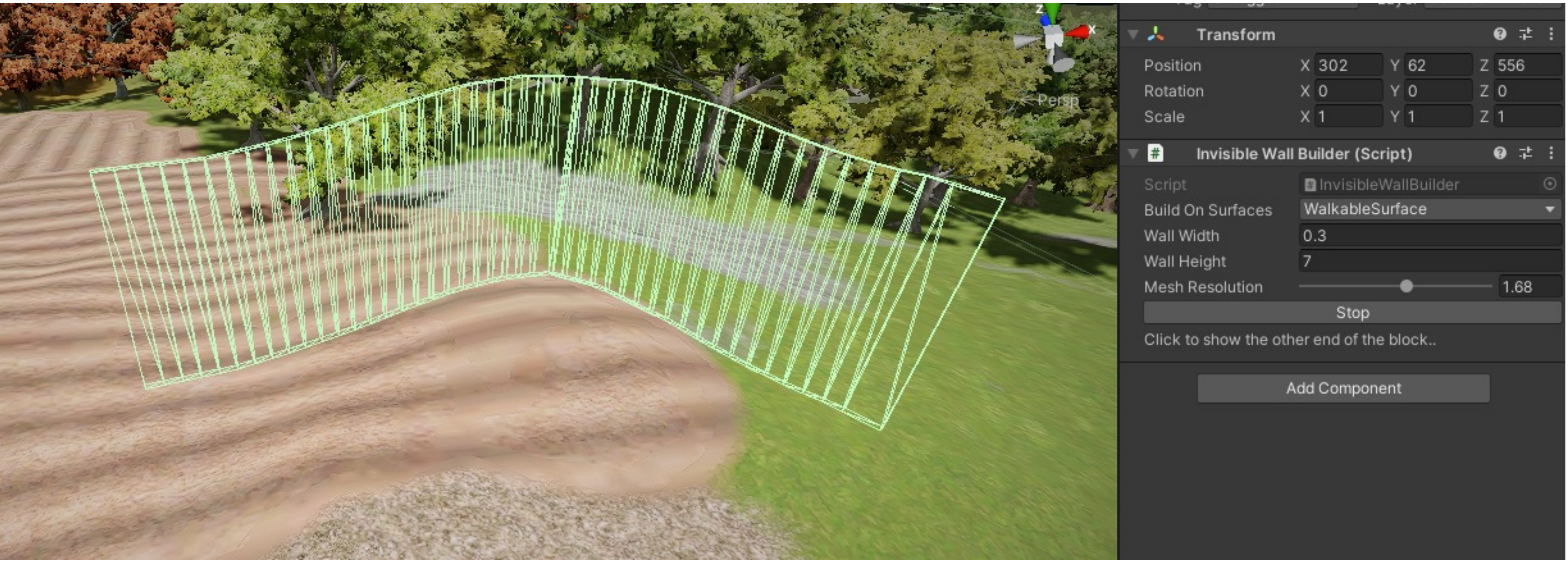
Click to indicate start + end of invisible walls (which exist as colliders only)

Raycast against a layer mask (I have set up WalkableSurface as a layer here)



Each piece of wall is a Game Object created as a child of the main object (the one with the InvisibleWallBuilder script)

With MeshResolution==0 it creates boxes


Transform

Position	X 302	Y 62	Z 556
Rotation	X 0	Y 0	Z 0
Scale	X 1	Y 1	Z 1

Invisible Wall Builder (Script)

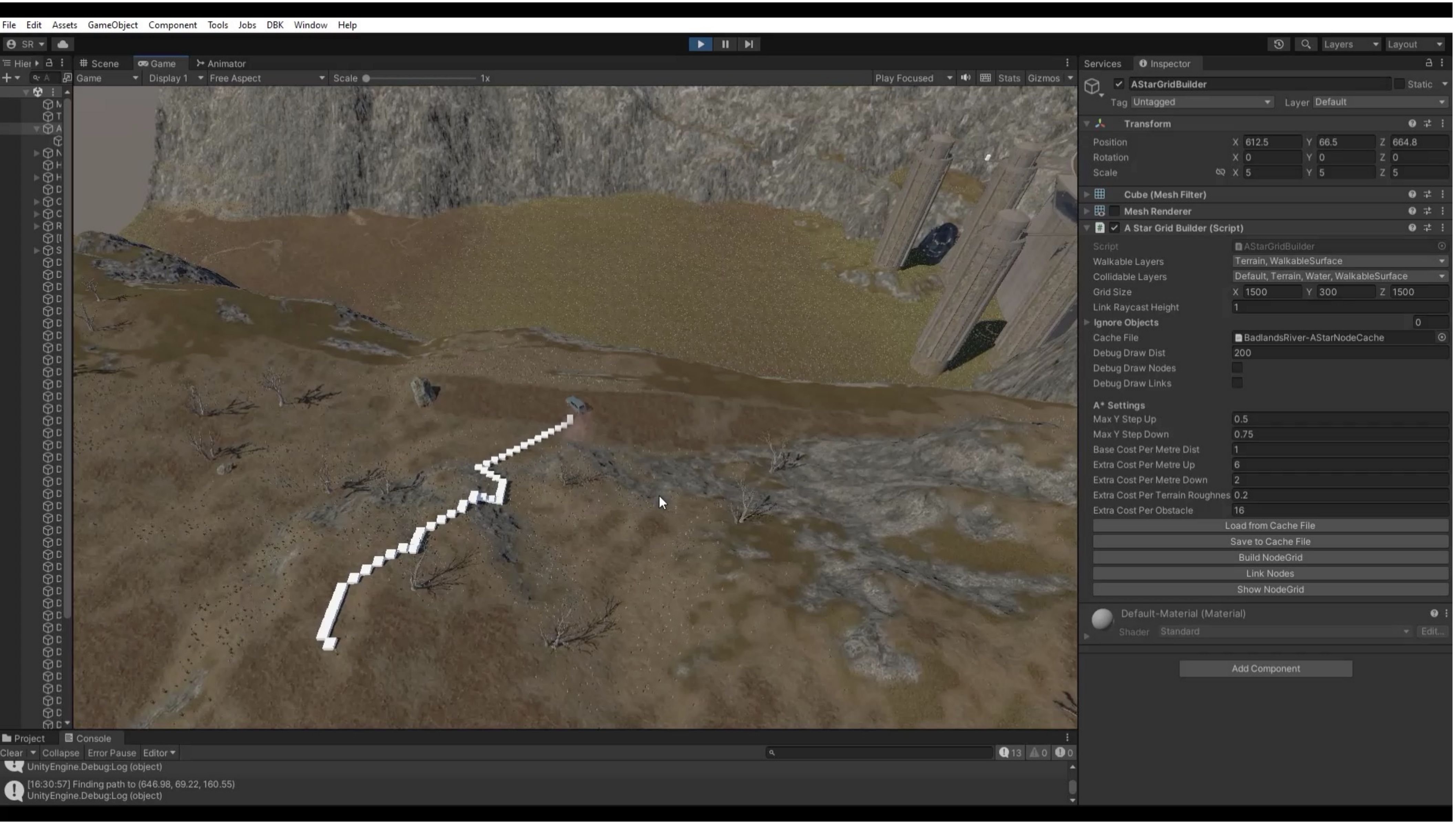
Script	InvisibleWallBuilder
Build On Surfaces	WalkableSurface
Wall Width	0.3
Wall Height	7
Mesh Resolution	1.68

Click to show the other end of the block..

With MeshResolution>0 it creates more complex collision meshes conforming to the surface

InvisibleWallBuilder code is in a separate PDF. This also illustrates how to programmatically build meshes as a set of triangles.

Pathfinding for Vehicles


File Edit Assets GameObject Component Tools Jobs DBK Window Help
Scene Game Animator
Game Display 1 Free Aspect Scale 1x
Play Focused Stats Gizmos
Services Inspector
Layers Layout
Project Console
Clear Collapse Error Pause Editor
Load from Cache File
Save to Cache File
Build NodeGrid
Link Nodes
Show NodeGrid
Add Component

AStarGridBuilder Static
Tag Untagged Layer Default

Transform

Position	X 612.5	Y 66.5	Z 664.8
Rotation	X 0	Y 0	Z 0
Scale	X 5	Y 5	Z 5

Cube (Mesh Filter)
Mesh Renderer
A Star Grid Builder (Script)

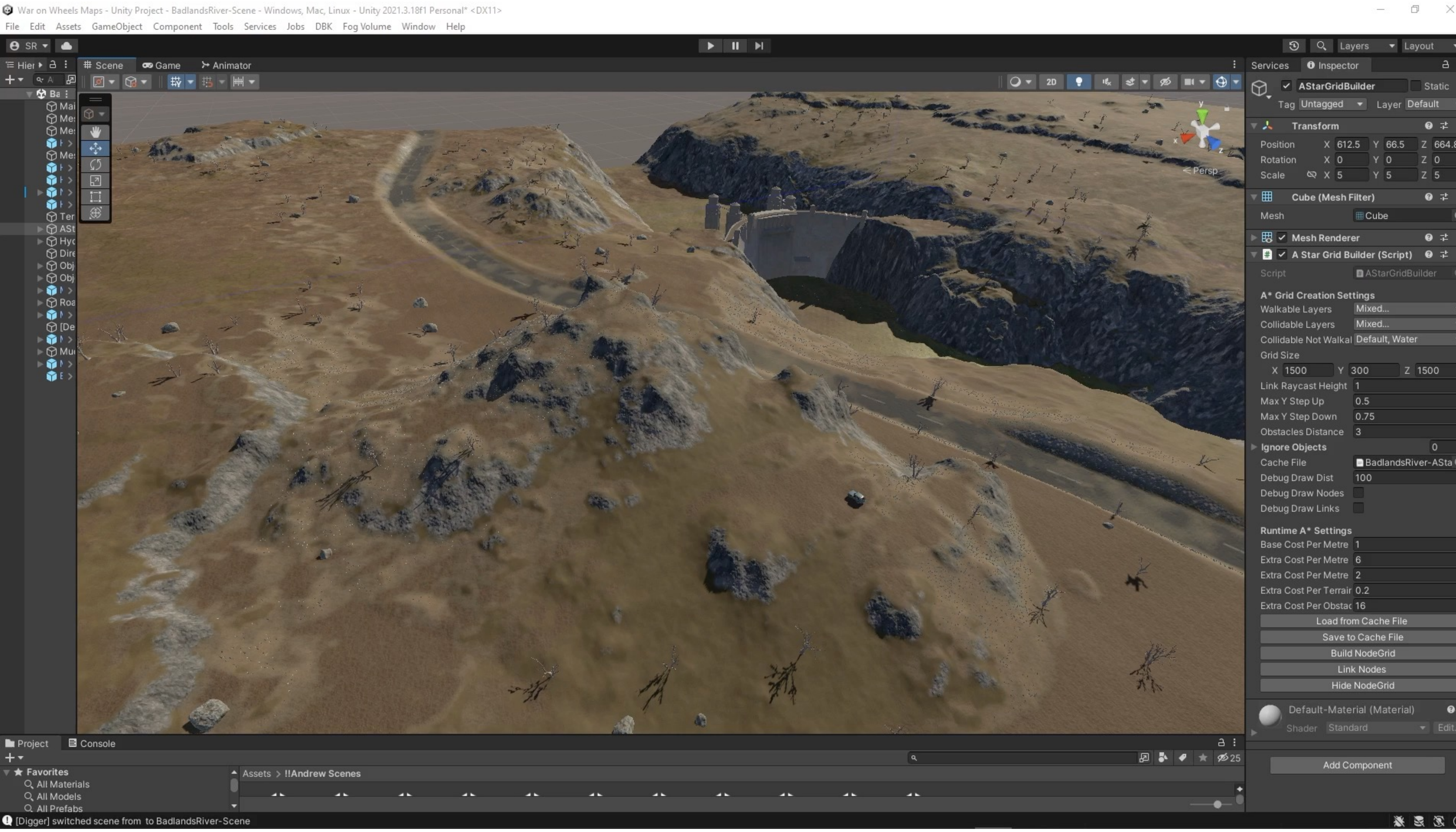
Script	AStarGridBuilder
Walkable Layers	Terrain, WalkableSurface
Collidable Layers	Default, Terrain, Water, WalkableSurface
Grid Size	X 1500 Y 300 Z 1500
Link Raycast Height	1
Ignore Objects	0
Cache File	BadlandsRiver-AStarNodeCache
Debug Draw Dist	200
Debug Draw Nodes	
Debug Draw Links	

A* Settings

Max Y Step Up	0.5
Max Y Step Down	0.75
Base Cost Per Metre Dist	1
Extra Cost Per Metre Up	6
Extra Cost Per Metre Down	2
Extra Cost Per Terrain Roughnes	0.2
Extra Cost Per Obstacle	16

Default-Material (Material)
Shader Standard

```
UnityEngine.Debug:Log (object)
[16:30:57] Finding path to (646.98, 69.22, 160.55)
UnityEngine.Debug:Log (object)
```



War on Wheels Maps - Unity Project - BadlandsRiver-Scene - Windows, Mac, Linux - Unity 2021.3.18f1 Personal* <DX11>

AStarGridBuilder Static
Tag Untagged Layer Default

Transform
Position X 612.5 Y 66.5 Z 664.8
Rotation X 0 Y 0 Z 0
Scale X 5 Y 5 Z 5

Cube (Mesh Filter)
Mesh Cube

Mesh Renderer

A Star Grid Builder (Script)
Script AStarGridBuilder

A* Grid Creation Settings
Walkable Layers Mixed...
Collidable Layers Mixed...
Collidable Not Walkal Default, Water
Grid Size
X 1500 Y 300 Z 1500
Link Raycast Height 1
Max Y Step Up 0.5
Max Y Step Down 0.75
Obstacles Distance 3
Ignore Objects 0
Cache File BadlandsRiver-ASta
Debug Draw Dist 100
Debug Draw Nodes
Debug Draw Links

Runtime A* Settings
Base Cost Per Metre 1
Extra Cost Per Metre 6
Extra Cost Per Metre 2
Extra Cost Per Terrair 0.2
Extra Cost Per Obstac 16

Default-Material (Material)
Shader Standard

[Digger] switched scene from to BadlandsRiver-Scene

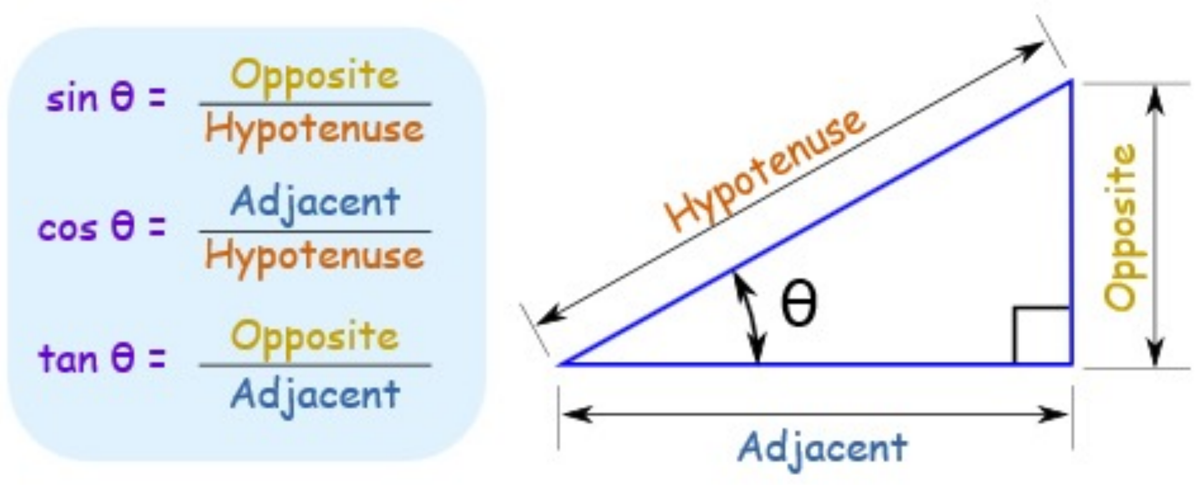
Fun with vectors
(basic operations which every game programmer should know)

Some Vector Operations

- Vectors are fundamental to several useful trigonometry/co-ordinate geometry operations; in Unity use Vector2 and Vector3 classes
- Vector3 has .x, .y., .z components and can represent direction + length, or perhaps just a 3D position
- Length (magnitude) of a vector is Mathf.Sqrt(v.x*v.x + v.y*v.y + v.z*v.z), (thanks Pythagoras!) or in Unity use v.magnitude
- Multiplying a vector by a scalar (float) simply multiplies .x, .y. and .z independently
- Adding two vectors simply adds their .x, .y., and .z independently, or in Unity use v3 = v1 + v2 (since the + operator has been overloaded for vectors)
- To get the difference between two vectors, subtract the .x, .y. and .z separately, or in Unity use vdiff = v1 - v2
- Length (magnitude) of a difference vector is distance between two points v1 and v2
- Normalized difference vector is the direction from one point to the other, e.g. to find a position p3 that is 25 units towards point p2 from point p1:
  - Vector3 diff = p2 - p1;
  - Vector3 dir = diff.normalized;
  - Vector3 p3 = p1 + dir * 25f;
- Length of a velocity vector is speed

Some Vector Operations

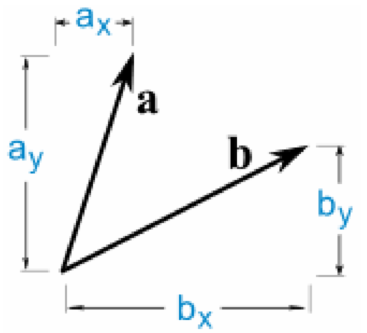
- In 2D games, Arctan(y, x) is also very useful (turns a direction vector into a rotation)
- In Unity:
  float rotation = Mathf.Atan2(float y, float x);


sin θ = Opposite / Hypotenuse
cos θ = Adjacent / Hypotenuse
tan θ = Opposite / Adjacent
Hypotenuse
θ
Opposite
Adjacent

- To turn a rotation angle into x,y components (i.e. the reverse of the above):
  - x = cos(ang)   e.g. cos(0)=1, cos(PI/2)=0, cos(PI)=-1
  - y = sin(ang)   e.g. sin(0)=0, sin(PI/2)=1, sin(PI)=0

See also: https://docs.unity3d.com/Manual/VectorCookbook.html

Dot Product



$\mathbf{a} \cdot \mathbf{b} = a_x \times b_x + a_y \times b_y$

So we multiply the x's, multiply the y's, then add.

The dot product of two normalised vectors

    float dot = Vector3.dot(vec1, vec2);

This is the cosine of the angle between the vectors

+1 means the two vectors are facing precisely the same way
-1 means they're facing precisely opposite
0 means they're perpendicular
+0.9 (for example) means they're facing "almost" the same way

So for example in a top-down 2D game, if we want to know if a zombie can see a player character (assuming the zombie has 180 degree field of view):

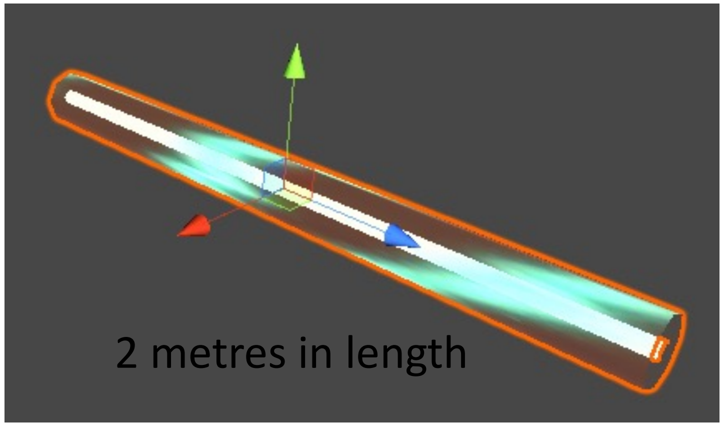
- Obtain normalised vectors (a) zombie's facing direction (in Unity, we can use zombie.transform.forward for this), and (b) direction from zombie to player (i.e. get the normalized difference in positions, as in last slide)
- If the dot product of (a) and (b) is greater than 0, the zombie can see the player
- (The next step is probably a raycast to see if the zombie's view of the player is blocked)

To get the actual angle between the vectors, use the arccosine of the dot product

    float ang = Mathf.Acos(dot);

Some Vector Ops: Example Demon Pit Lasso sections (this code is in Player.cs)

```
 void FixedUpdate() {
   if (!isTeleporting && targetedTeleportNode!=null &&
Input.GetMouseButton(1)) {
        isTeleporting = true;
        rigid.isKinematic = true;
        rigid.useGravity = false;
        Vector3 myPos = transform.position;
        Vector3 teleportPos = targetedTeleportNode.transform.position;
        Vector3 diff = teleportPos-myPos;
        initialDistanceToTeleportNode = diff.magnitude;

        // spawn the lasso sections
        diff = teleportPos-myPos;
        Vector3 directionToTeleportNode = diff.normalized;

        int numSections = Mathf.FloorToInt(initialDistanceToTeleportNode/2f)+1;
        float lengthOfAllSections = numSections * 2f;
        float offsetDist = lengthOfAllSections-diff.magnitude;
        for (int i=0; i<numSections; i++) {
           GameObject go = Instantiate(lassoSectionPrefab);
           go.transform.position = myPos + (i*2f-offsetDist)* directionToTeleportNode;
           go.transform.LookAt(teleportPos);
           activeLassoSections.AddFirst(go);
        }

}
```



<u>Class members:</u>
activeLassoSections is a List<GameObject>
initialDistanceToTeleportNode is a float
targetedTeleportNode is a GameObject

```
 if (targetedTeleportNode!=null && isTeleporting) {
      // move player rapidly towards teleport node
      Vector3 diff = (targetedTeleportNode.transform.position - transform.position);
      float dist = diff.magnitude;
      if (dist>1f) {
          Vector3 dir = diff.normalized;
          rigid.MovePosition(transform.position + dir*Time.fixedDeltaTime*50f*(1f+dist/30f));
          float fracDistToGo = dist/initialDistanceToTeleportNode;
          if (fracDistToGo>0.5f)
              fpsCamera.fieldOfView = Mathf.Lerp(60f, 90f, 2f*(1f-fracDistToGo));
          else
              fpsCamera.fieldOfView = Mathf.Lerp(90f, 60f, 2f*(0.5f-fracDistToGo));
      }
      else {
          // finished
          rigid.isKinematic = false;
          rigid.useGravity = true;
          isTeleporting = false;
          fpsCamera.fieldOfView = 60f;
          foreach(GameObject go in activeLassoSections)
              Destroy(go);
          activeLassoSections.Clear();
      }
  }

} // end of FixedUpdate()
```

Some Vector Ops: Examples Controlling a homing missile (this code is in Projectile.cs)

```
void FixedUpdate() {
    if (hasHoming && isAlive) {
        Vector3 pos = transform.position;
        Vector3 vel = rigid.velocity;
        Vector3 forward = vel.normalized;

        if (homingTarget==null || homingTarget.isDead) { // lock on ?
            homingTarget = null;
            float closestDist = HOMING_DIST*10f;
            for (int i=0; i<MonsterManager.allActiveMonsters.Count; i++) {
                Vector3 diff = MonsterManager.allActiveMonsters[i].transform.position - pos;
                float dist = diff.magnitude;
                if (dist <= HOMING_DIST) {
                    Vector3 dir = diff.normalized;
                    float dot = Vector3.Dot(forward, dir);
                    if (dot>0.9f) {
                        if (dist<closestDist) {
                            closestDist = dist;
                            homingTarget = MonsterManager.allActiveMonsters[i];
                        }
                    }
                }
            }
        }

        if (homingTarget!=null) {
            Vector3 targPos = homingTarget.gameObject.transform.position;
            if (homingTarget.isWalker)
                targPos.y += 0.9f; // don't aim for their feet!
            Vector3 turnToFacing = (targPos-pos).normalized;
            Vector3 newFacing = Vector3.RotateTowards(forward, turnToFacing, Time.fixedDeltaTime*homingStrength, 1f);
            rigid.velocity = newFacing * vel.magnitude;
            if (homingTurnFacing)
                transform.forward = newFacing;
        }
    }
}
```

homingTarget is a Monster object

MonsterManager.allActiveMonsters is a List<> of Monster objects

HOMING_DIST is a float

Vector3 result = Vector3.RotateTowards(Vector3 current, Vector3 target, float maxRadiansDelta, float maxMagnitudeDelta);

Some Vector Ops: Examples Part of the Explosion logic. (this code is in Projectile.cs)

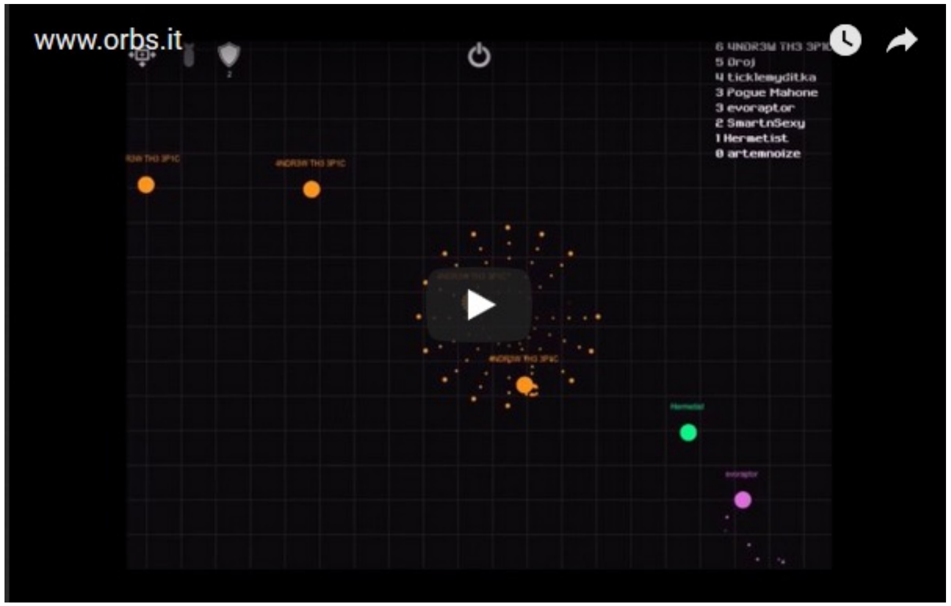
```
public void RemoveWithParticleFx(Vector3 explosionPoint, GameObject particlePrefab) {
        Destroy( this.gameObject );

        GameObject go = Instantiate(particlePrefab);
        Vector3 diff = Player.currPlayerPos - explosionPoint;

        if (doExplosionPositionOffsets) { // doExplosionPositionOffsets is a public Boolean (inspector setting)
            float moveDist = 2.5f;
            if (diff.magnitude<3f)
                moveDist = diff.magnitude - 0.5f;
            // move the particle emitter closer to the player to stop ugly clipping with geometry
            go.transform.position = explosionPoint + (diff.normalized*moveDist);
        }
        else
            go.transform.position = explosionPoint;

        if (explosionRadius>0f && explosionDamage>0f) {
            // Physics.OverlapSphere queries the physics world to get an array of colliders inside a sphere region
            // The results are stored in the array collResults which is predefined to avoid garbage
            int hits = Physics.OverlapSphereNonAlloc(explosionPoint, explosionRadius, collResults,
GameManager.explosionDamageLayerMask); // this layer mask will select only Player or Monster objects
            for (int i=0; i<hits; i++) {
                go = collResults[i].gameObject;
                if (go.layer==GameManager.layerPlayer)
                    go.GetComponent<Player>().OnHitByExplosion(explosionDamage, explosionPoint);
                else
                    go.GetComponent<Monster>().OnHitByExplosion((int)explosionDamage, explosionPoint);
            }
        }
}
```

Some useful game programming patterns and efficiency considerations

Binning (Bucketing)

- Storage of data elements in small sets, separated by (typically) their region in space
- Each small set may be stored as a simple array or linked list
- The high-level structure is essentially a 2D or 3D array, indexed on world position rounded off to some predefined size (e.g. 5m x 5m x 5m) – each array entry is a Bin (or Bucket)
- As objects move, they are added to/removed from bins as appropriate
- Only a small set has to be traversed when we need to identify interactions between moving objects
- This is certainly how Unity's physics engine also operates, in order to know which objects are close to each other, prior to performing more expensive collision calculations
- E.g. for bullets in my "Orbs.it" web-game:
    - Orbs are registered to bins
    - Bullets only consider orbs in the same bin
    - See also:

http://www.psychicsoftware.com/category/orbs-it/

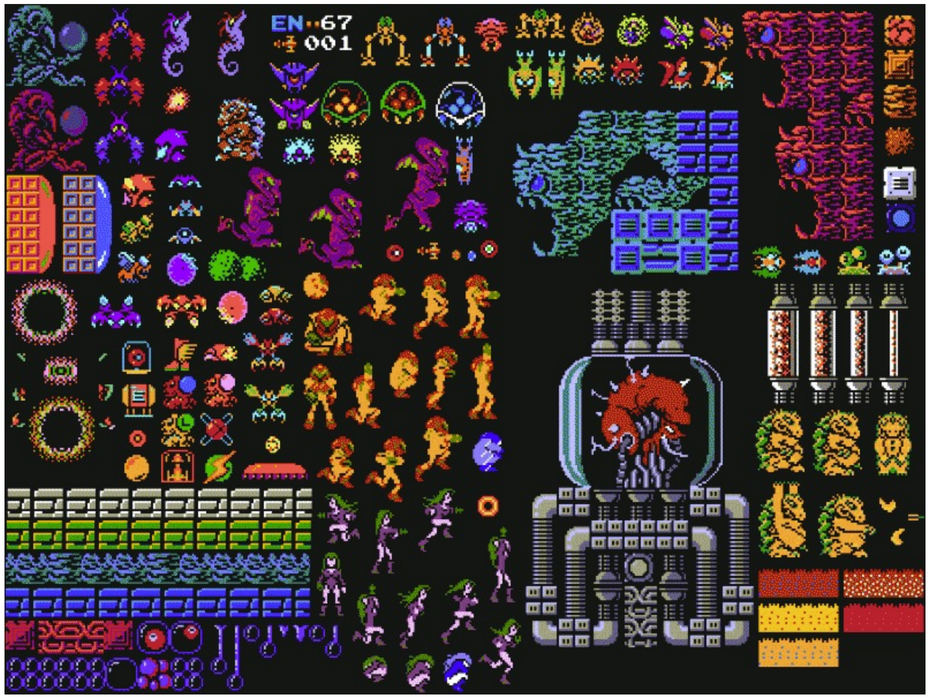


The Matterjs collision detector Demonstrates coarse-to-fine algorithms

- Matterjs is a Javascript (i.e. open source) physics engine for 2D web-games (see http://brm.io/matter-js/)
- Its collision detector is an interesting case study in efficiency for realtime systems
- How to efficiently and accurately find collisions between pairs of objects when there may be thousands of objects, and it's expensive to calculate collisions on pairs of objects if they have complex shapes?
- Three steps (coarse-to-fine, aka broad-to-narrow)
- **Broadphase (coarse stage)**
  - As objects move, they are registered/de-registered from 'buckets' that they enter/leave (default bucket size in matterjs is 48x48, and a grid of buckets covers the world)
  - Only consider objects for collision if they are registered to the **same bucket**
  - Objects may be in multiple buckets at the same time (i.e. any bucket that their bounding box overlaps)
- **Midphase**
  - If objects are in the same bucket, then do their **axis-aligned bounding boxes** overlap?
- **Narrow phase (fine stage)**
  - If both broadphase and midphase have been passed for a pair of objects, we can now consider their **actual vertices** to look for penetration. This will also yield precise point(s) of contact and surface normals at those points

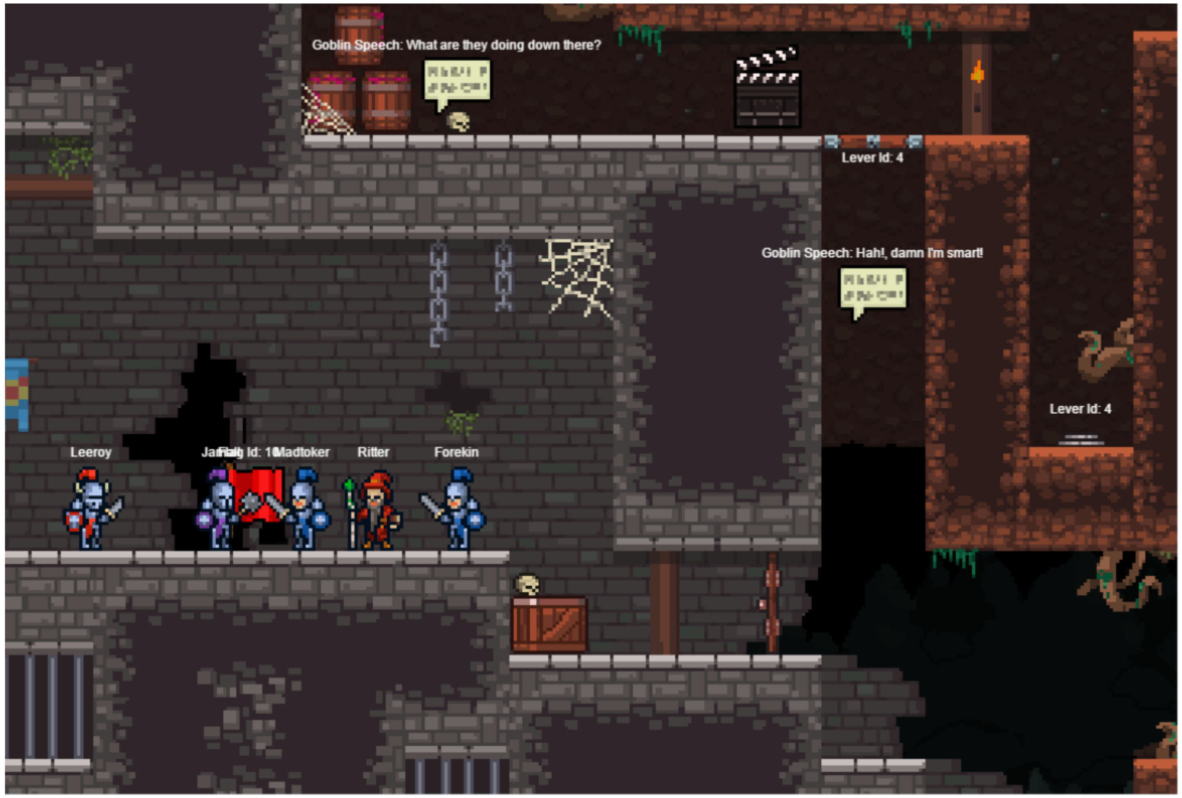
Texture Atlases

- Large images containing a collection, or "atlas", of sub-images, each of which is a texture map for some part of a 2D or 3D model.
- The sub-textures can be rendered by modifying the texture coordinates of the object's uvmap on the atlas, essentially telling it which part of the image its texture is in (cookie cutter analogy)
- In an application where many small textures are used frequently, it is often more efficient to store the textures in a texture atlas which is treated as a single unit by the graphics hardware
  - ideally a single draw call
  - certainly a reduction in draw-state changes in graphics API



Triggers (sensors)

- Invisible components that are activated (triggered) when a character or other object passes inside them
- Very common (and useful) mechanism in games
- In Unity, use a Collider with its Trigger property checked
- Various things they're used for in Goblins & Grottos:
  - Goblin speech
  - Cutscene triggers
  - Trap triggers
  - Level exit (finished)
  - Etc.


Goblin Speech: What are they doing down there?
Lever Id: 4
Goblin Speech: Hah!, damn I'm smart!
Lever Id: 4
Leeroy
Madtoker
Ritter
Forekin

Various things I have written about G&G:
http://www.psychicsoftware.com/category/goblins-grottos/

Runtime Efficiency (1/2)

- Spread out work so that it tends to happen on different frames (a single very CPU-intensive frame could cause a visible stutter in framerate)
  - Avoid putting code in Update() unless you *really* need it to happen every frame
  - Perhaps iterate a list of jobs that need to be done, processing just 1 of them per frame
  - Use Coroutines / Invoke calls instead, to make work happen at less frequent intervals, or even to break large pieces of work over multiple frames
  - Avoid putting code in OnCollisionStay() unless you *really* need to process it every frame (otherwise use OnCollisionEnter())
- One useful trick for visual effects that are optional to your game is to monitor Time.deltaTime to see whether to deploy special effects, or decide whether unimportant game objects should be culled.. i.e. is the player's device struggling to achieve a good framerate?

Runtime Efficiency (2/2)

- Know your language's memory management and garbage collector!
  - Different languages/runtimes have different approaches
  - Know what is stored in the **Stack** and what is stored in **Heap** memory
  - In C#, generally primitive data (int, float, etc.) and also structs are stored in the Stack, while objects (including strings) are stored in the Heap
  - Stack data is local to a function and is released as soon as the function ends. Heap data is longer-lived and will wait to be garbage-collected
  - In C#, the GC approach is 'mark and sweep’
- Use non-alloc raycasts in Unity rather than obtaining a new array of results each time (because arrays use the Heap)
- Use Object Pools (see below)
- Perhaps use multi-threading, or at least Coroutines (see 'Thread Ninja' asset in the Unity Assetstore)

Unity's (Horrible) Memory Management (1/2)
aka "Strings are your enemy"

- https://unitygem.wordpress.com/memory-management/
- https://blogs.msdn.microsoft.com/abhinaba/2009/01/30/back-to-basics-mark-and-sweep-garbage-collection/
- https://unity3d.com/learn/tutorials/topics/performance-optimization/optimizing-garbage-collection-unity-games
- When a new Heap object is requested, the following steps take place:
  - First, Unity must check if there is enough free memory in the heap. If there is enough free memory in the heap, the memory for the object is allocated.
  - If there is not enough free memory in the Heap (or if the Heap is fragmented as cannot provide a contiguous block of memory that is large enough), Unity triggers the garbage collector in an attempt to free up unused heap memory. This can be a slow operation. If there is now enough free memory in the heap, the memory for the variable is allocated.
  - If there isn't enough free memory in the heap after garbage collection, Unity increases the amount of memory in the Heap. This can be a slow operation. The memory for the variable is then allocated.
- Strings use the Heap, and are *immutable.* Use StringBuilder where possible.

(2/2)

- Garbage Collector needs to check for the *reachability* of all objects allocated on the heap – this means a check for whether they can be accessed from any code that is or could be running – if they can be then the object is retained otherwise it is released
- "Mark": start from root/global references, and recursively visit all reachable objects, marking them as accessible
- "Sweep": pass through *all* objects and delete those not marked as accessible
- No other processing can happen during the mark-and-sweep process, hence framerate stutters can happen in realtime environments/games
    - And players will hate you and mock your dev skills
- Minimise GC calls, and minimise the impact of them when they do happen, by minimising the amount of objects being instantiated and discarded
- See also: https://docs.unity3d.com/Manual/BestPracticeUnderstandingPerformanceInUnity4-1.html
- Modern generational garbage collectors improve things a lot: the partition of objects into different *generations* (time intervals) based on time of allocation, and giving them different GC policies depending on age (older generations are rarely tested as part of the mark & sweep process). Unity does not use a modern garbage collector.. but they keep promising to

The Object Pool pattern

- Instantiating large numbers of objects at run-time can be CPU expensive (esp. if they're complex and include sprites/meshes etc.)
- Deleting them can also be expensive, and subsequent garbage collection may cause frame stutters
- The Object Pool pattern is used in games and other realtime systems:
    - Stores a pool (data structure, e.g. List<>) of 'spare/unused' copies of objects, currently inactive and unused, yet still instantiated in memory
    - At run-time, objects are taken from and returned to the pool, rather than actually instantiating and deleting them from memory
- See my PoolManager class.. next two slides

```
public class PoolManager {

    // Each dictionary entry is indexed by Prefab name
    // Each LinkedList contains spare objects of that type
private static Dictionary<string,LinkedList<GameObject>> Pools = new
Dictionary<string, LinkedList<GameObject>>();

    public static GameObject GetGameObject(string prefabName) {
        if (!Pools.ContainsKey(prefabName))
            Pools.Add(prefabName, new LinkedList<GameObject>());

        LinkedList<GameObject> pool = Pools[prefabName];

        GameObject go;
        if (pool.Count>0) {
            go = pool.First.Value;
            go.SetActive(true);
            pool.RemoveFirst();
        }
        else {
            go = (GameObject)GameObject.Instantiate(Resources.Load(prefabName));
            go.name = prefabName; // get rid of "(Clone)"
        }

        return go;
    }
    public static void ReturnGameObject(GameObject go) {
        if (!Pools.ContainsKey(go.name))
           Pools.Add(go.name, new LinkedList<GameObject>());

        LinkedList<GameObject> pool = Pools[go.name];

        pool.AddFirst(go);

        go.transform.position = Vector3.zero;

        if (go.transform.rotation.z!=0f) {
           go.transform.rotation = Quaternion.identity;
        }
        go.SetActive(false);
    }

}
```

- Often, implementations of the Object Pool pattern start by ‘pre-warming’ the pools by instantiating (and deactivating) a bunch of objects when the game starts
- I haven’t found this necessary to do in any of my games – I’m simply trying to avoid garbage

Unity non-alloc Physics queries

- Recall the use of Physics.RaycastAll, Physics.OverlapBox, Physics.OverlapSphere etc. for querying spatial interactions with colliders in the game
- These methods return new arrays of RaycastHit structs… so if this is done often, there can be a lot of overhead
- If you will need to make these tests often (i.e. multiple times per second.. e.g. in a platformer game) you should use the "non-alloc" versions:
  - Physics.RaycastNonAlloc
  - Physics.OverlapBoxNonAlloc
  - Physics.OverlapSphereNonAlloc
- These methods receive an array as an argument, and they populate the array with the results
- So you could, for example, create a static array and send that each time the test was needed. The same array would then be re-used each time without **new** or **delete** operations happening

Pre-calculated lookup tables (1/2)

In games (and other real-time systems) it's often useful to create custom structures of 'lookup' information when the program or game level starts, to reduce CPU load per-frame at run-time

E.g. snow in Goblins & Grottos:

Map load-time:

- Perform vertical raycasts downwards from top of the world, at horizontal intervals of 32 pixels, to find collision with highest physics body.
- Store in a lookup array indexed by: Math.floor(x/32)

Run-time:

- When we need a new snow particle, pick a random position just above the player's current camera position in the game world
- Only create a particle if this position is higher than the lookup array at that horizontal position (i.e. no snow underground!)
- As snow particles are updated, query the internal Matter.js buckets for objects whose bounding box contains the particle position – but only if the snow particle is currently below the value stored in the lookup array at its horizontal position.
  - Rather than having the snow particles under 'proper' control of the physics engine (since that would be too expensive for 100s of snowflakes)

Pre-calculated lookup tables (2/2)

Water/Lava in G&G

- This is simply a 2D lookup array (bins), indexed by Math.floor(x/32) and Math.floor(y/32), containing the 'liquid type' at each position: 0=none, 1=water, 2=lava
- Moving objects periodically test this lookup array to see if they have just entered/left a liquid
- Most objects only test once per 300ms; fast-moving ones test every frame; sleeping objects don't test at all

Navmaps in G&G

- Navigation map: stores valid positions for AI-controlled wizards/witches to teleport to
- Valid positions are directly above static ground/platform objects (in a 32x32 pixel grid).
- At map load-time, iterate through all static physics bodies. For each: check whether or not there is a static physics body directly above them
- Stored in a 1D array containing x,y positions
- At run-time, when a wizard/witch wants to teleport:
  - Pick randomly from the array, until a position that's close enough is found
  - Cast a ray downwards from just above this position, to decide the precise position to move the wizard/witch to (if a non-static object is currently in the way, the character will therefore teleport on top of it rather than inside it)

Scoping

- What is relevant for processing?
    - Physics objects only if moving now or recently (physics engines normally look after this with the concept of 'sleeping' or 'idling')
    - Physics and other objects only if onscreen (or other specific times) – in Unity you could use the Renderer.isVisible property, or MonoBehaviour.OnBecameVisible() and MonoBehaviour.OnBecameInvisible() to control whether scripts are running
    - Examples from G&G:
        - enemy AI code only executes if the character is onscreen (or 'nearly' onscreen)
        - I made some modifications to matterjs internal code (thank you, open-source!) so that offscreen physics bodies are not normally processed
            - Unless they are moving fast
            - Or unless they have been specifically flagged to 'always process until time X' – e.g. if an offscreen trapdoor opens then all nearby non-static physics bodies will be flagged to "always process until now+3000ms")
        - These specific use-cases were selected by studying the Chrome profiler for CPU bottlenecks while the game runs
- Scoping can be very relevant for networked games with lots of moving objects. At what rate do we need to send update network packets to the other players? - might depend on how close the object is to them? (also consider zone-based structure for MMO server clusters)