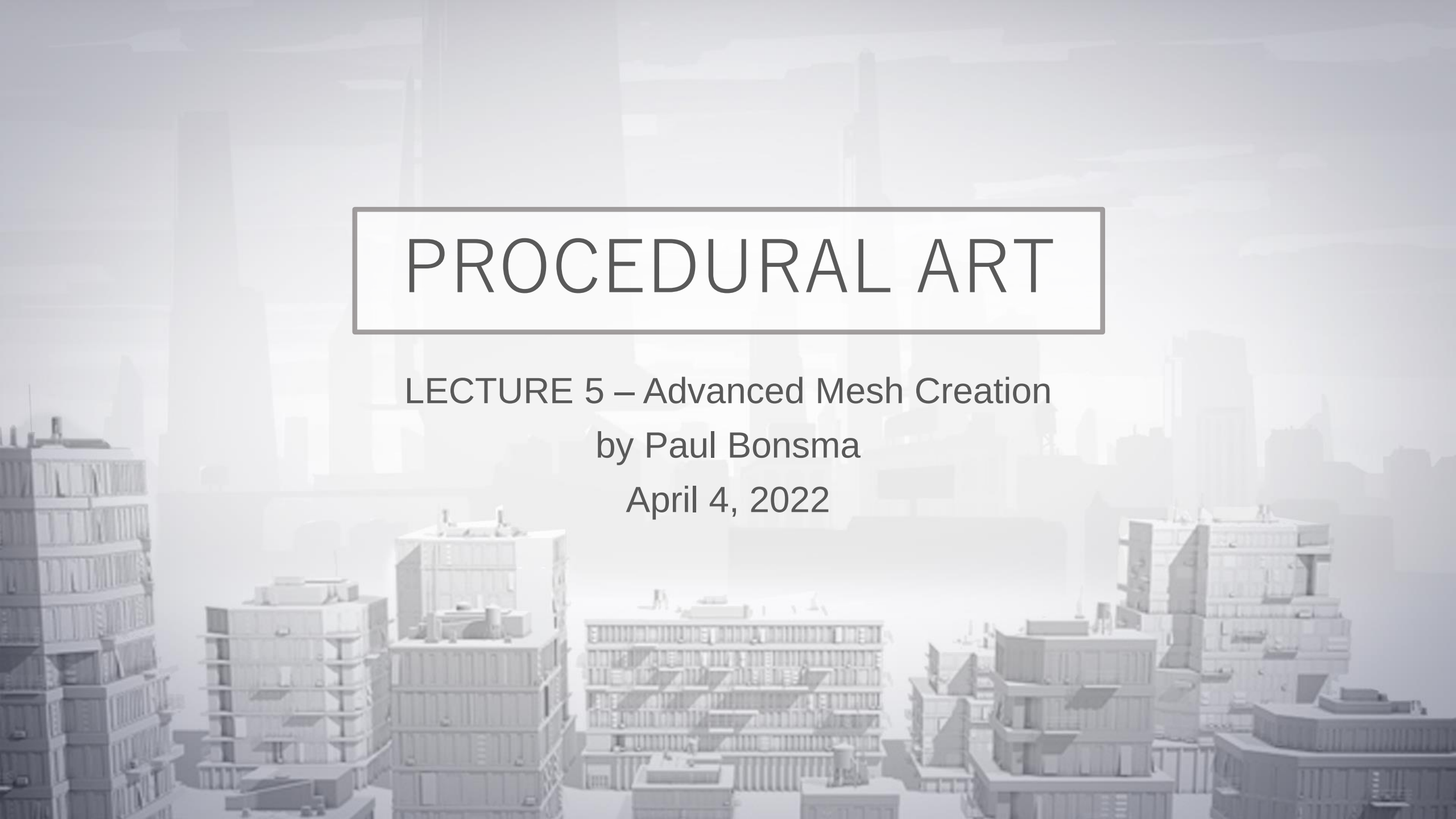


# PROCEDURAL ART

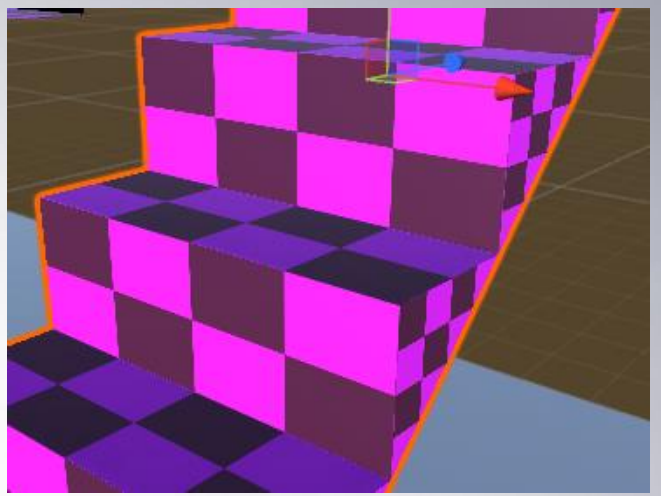
LECTURE 5 – Advanced Mesh Creation

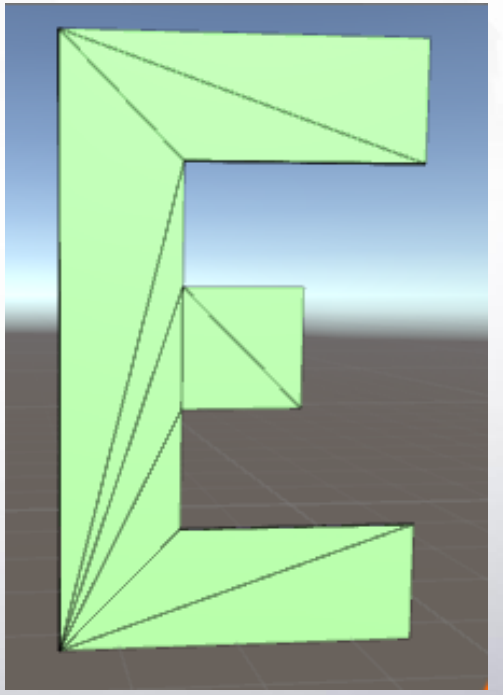
by Paul Bonsma

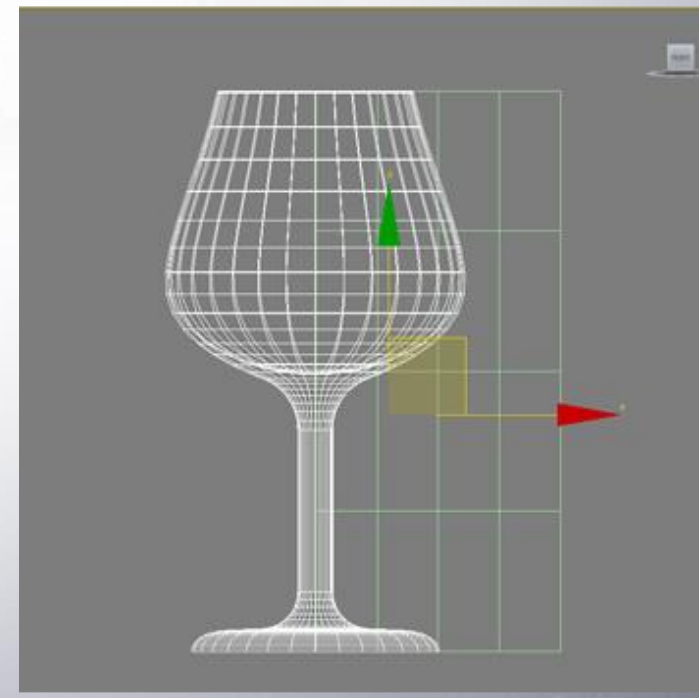
April 4, 2022

# Last Week

- Mesh basics: vertices, triangles, uvs, normals
- MeshBuilder class
- Unity editor tools: CurveEditor
- Mesh creation methods:
  - Lathe
  - Extrude (triangulate)

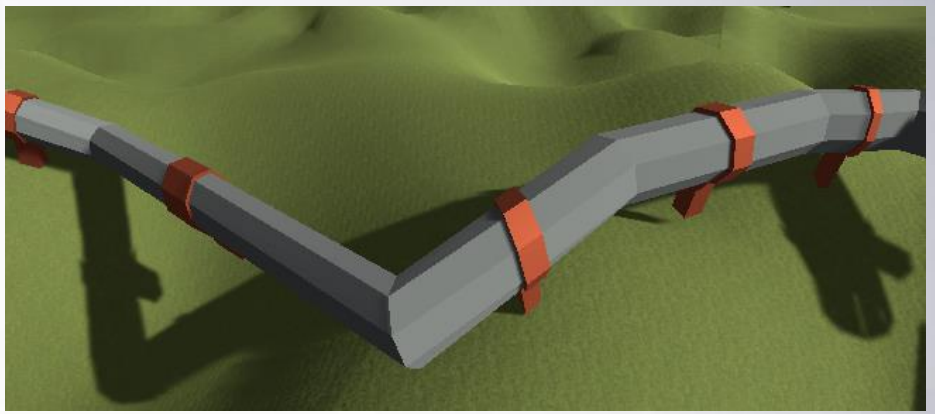




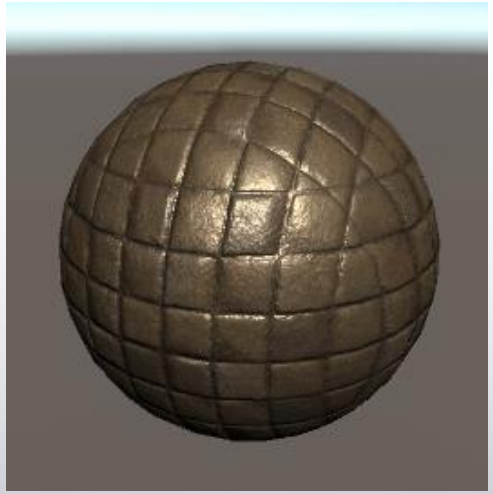


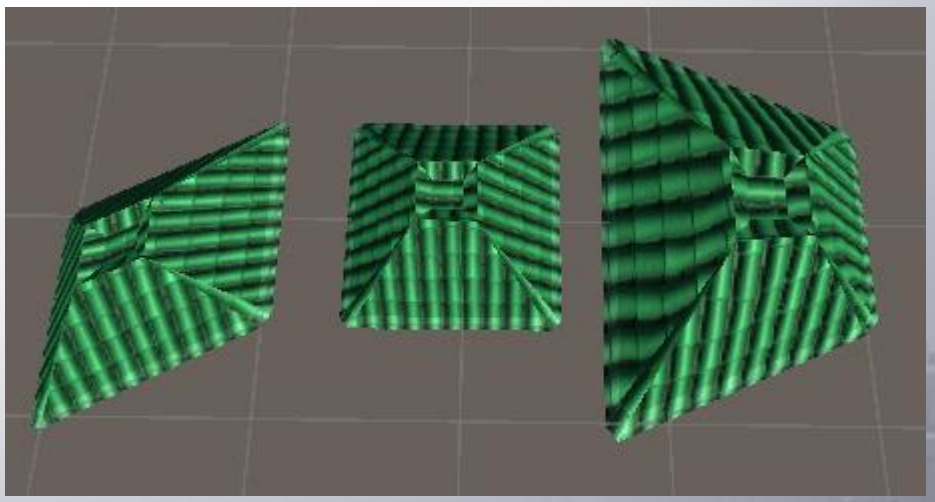
# This Week

- Mesh modification method examples:
  - Warp quad
  - Warp curve



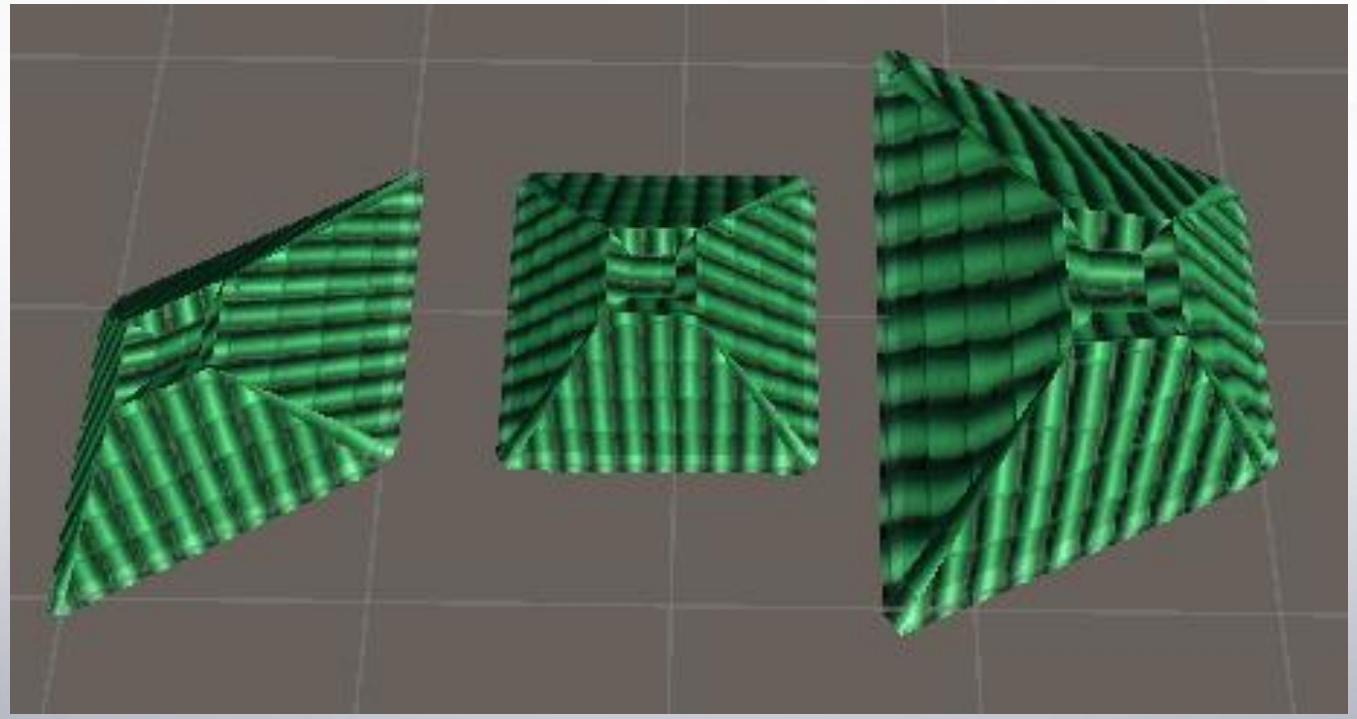
- Texturing meshes:
  - Automatic uv assignment
  - Shader solutions (3d coordinates, triplanar texturing)





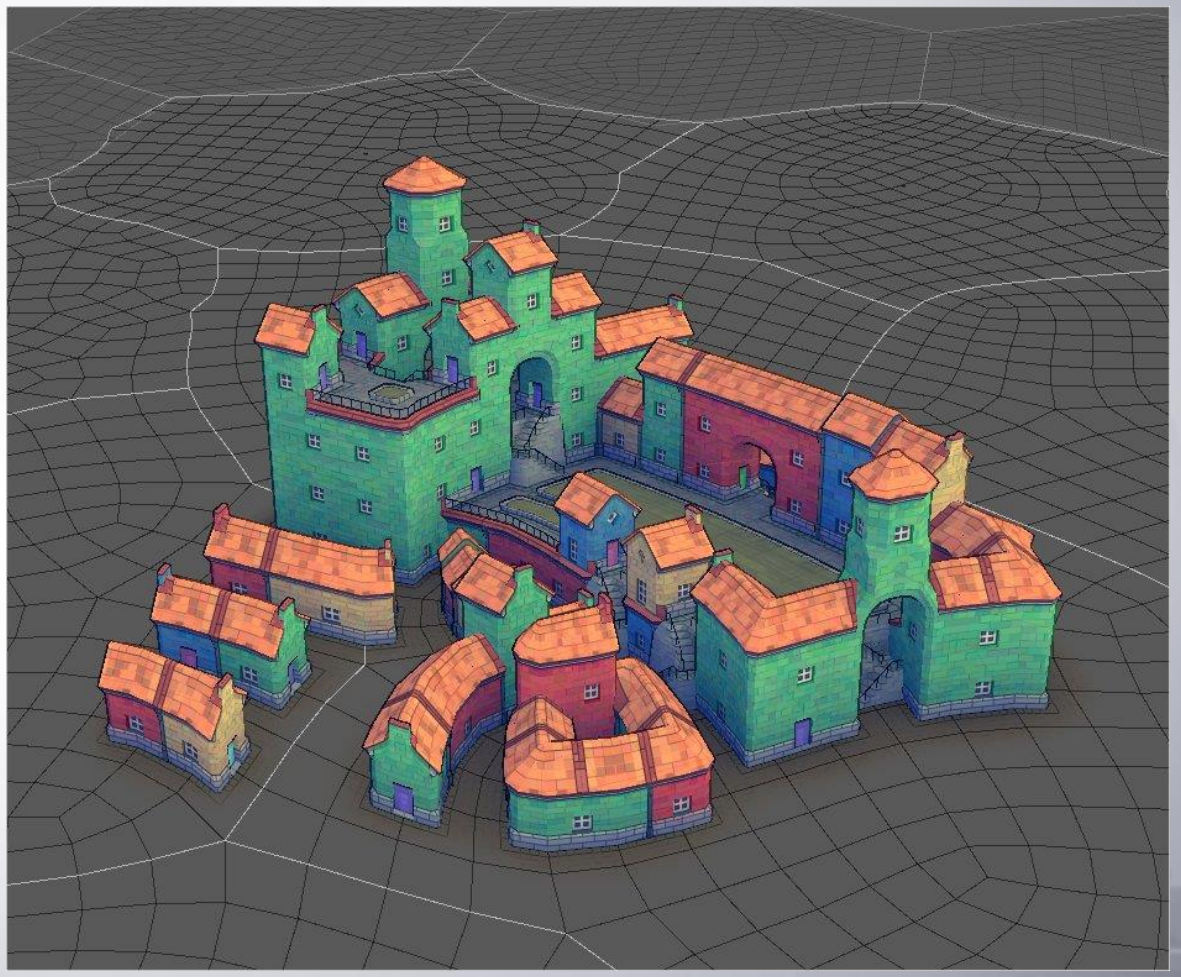
- Assets & scenes:
  - sharedMesh
  - OBJ files

# Warping Meshes: Quad



# Townscaper

- Recall the townscaper example from last week
- Key technique: slightly distort or *warp* mesh modules such that they fit on a distorted grid
- How can we do this?
- We assume that:
  - We have regular block shaped modules (like the Kenney assets)
  - We know the four corners of the ground quad

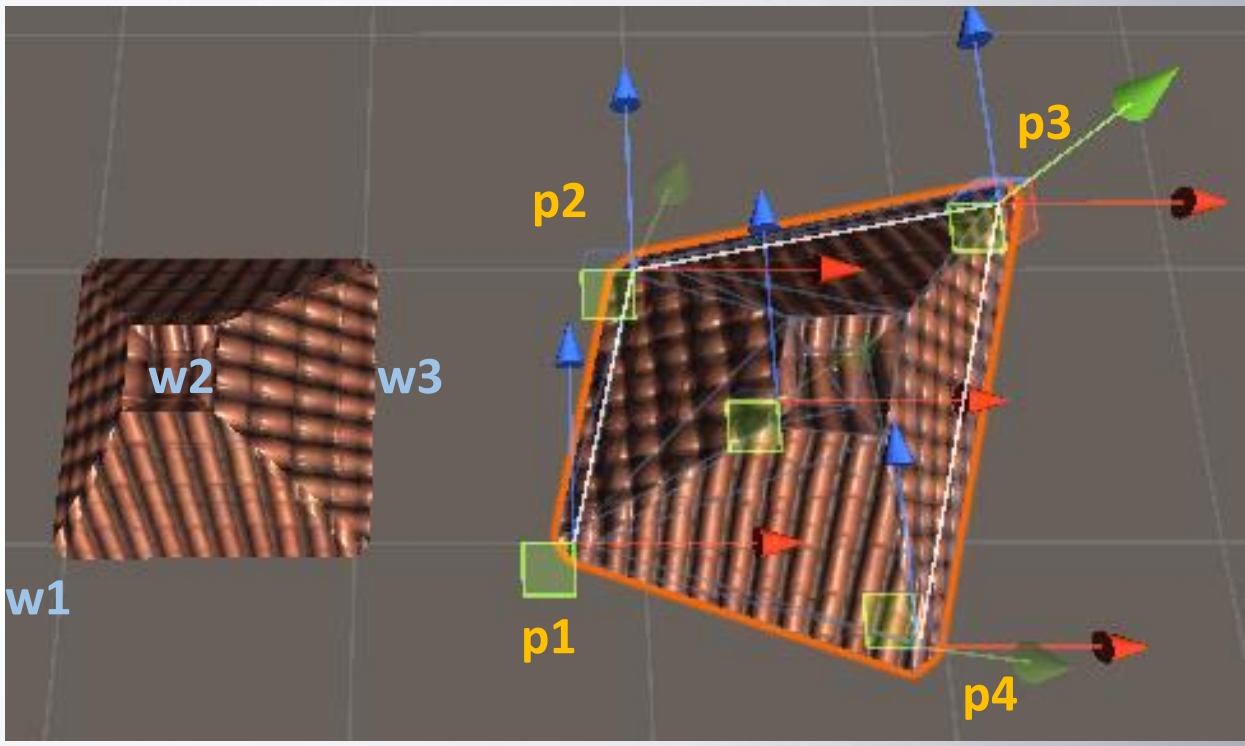


# Affine Combinations

- We have four (quad) corner points p1..p4 in world space, and we want to express points of our rectangular input mesh as a combination of those.
- To be precise:
- *A point v is an affine combination of points $p_1 \ldots p_n$ if we can find coefficients $c_1..c_n$ such that:*

$$v = c_1 * p_1 + \ldots + c_n * p_n$$
$$\text{where } c_1 + \ldots + c_n = 1$$

- For instance:
  - Roof corner *w1:  c1=1, c2=0, c3=0, c4=0*
  - Roof center *w2:  c1=c2=c3=c4=0.25*
  - Middle of right side *w3:  c1=c2=0,  c3=c4=0.5*



# Aside

- A *linear combination* is the same, but without the sum = 1 condition:
  $v = c_1 * p_1 + \ldots + c_n * p_n$
- A *convex combination* is the same as affine, but adds the condition that every coefficient is between 0 and 1:
  $v = c_1 * p_1 + \ldots + c_n * p_n$
  *w*here $c_1 + \ldots + c_n = 1$ and for all $i$: $0 \leq c_i \leq 1$

*Geometric meaning / intuition:*

- *Linear combinations* make sense when viewing vectors as directions / basis vectors (see 3D Math)
- *Affine combinations* and *convex combinations* make sense when viewing vectors as positions.
  - With affine combinations you can get every point in the minimum line / plane / .. that contains all input points
  - With convex combinations you can get every point inside the *convex hull* spanned by the points (line segment, triangle, tretrahedron, …)

# Finding the Coefficients

- So, we know that we need to use affine combinations of four corner points to map input vertices to (warped) output vertices.
- …but how do we find the coefficients c1…c4 for each input point?

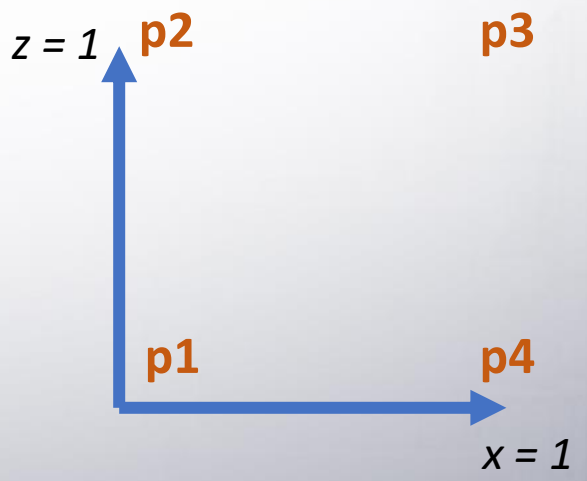
- Method:
  - We use the *x* and *z* coordinates of the input point (ignoring the *y* coordinate)
  - Map them both to the range 0…1 using multiplication and addition (for the Kenney fantasy town assets: just add 0.5 to both)
  - Then we use the formula:

    $c1 = (1-x) * (1-z)$

    $c2 = (1-x) * z$

    $c3 = x * z$

    $c4 = x * (1-z)$
- You can verify that c1+c2+c3+c4 = 1!
- Note that x=0,z=0 maps to p1, x=0,z=1 maps to p2, etc.

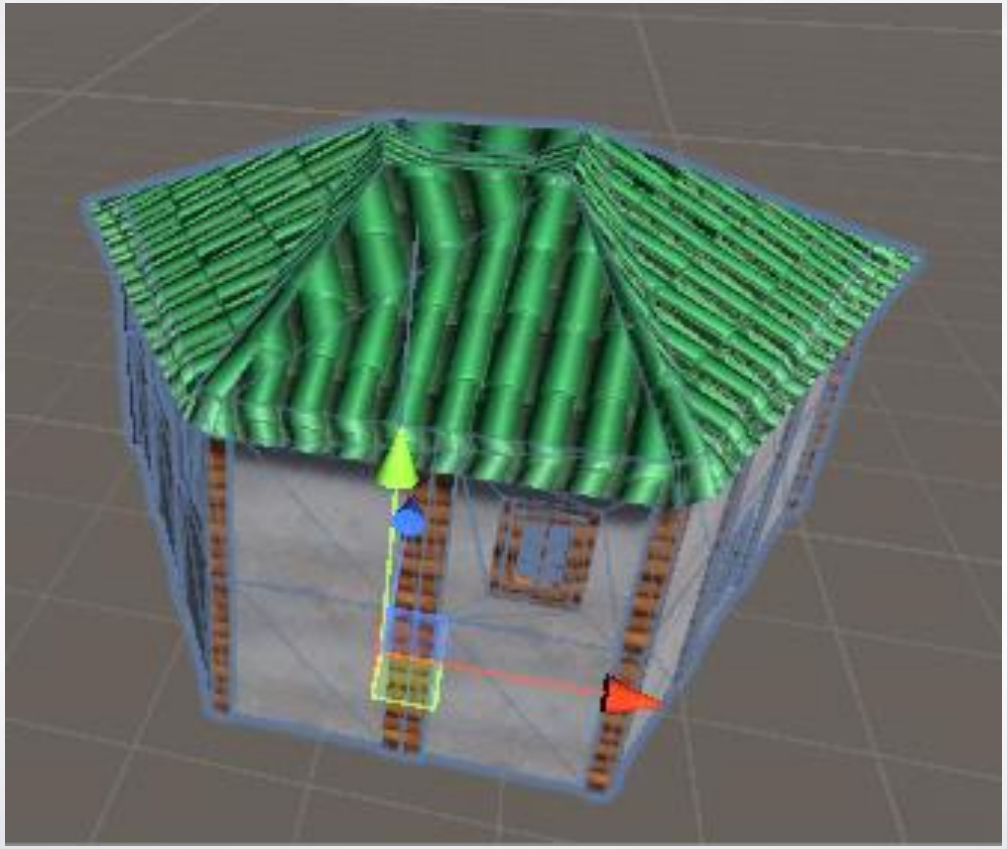


# Unity Implementation

- Create a new script that inherits from *MeshCreator.* So it needs to implement the *RecalculateMesh* method.
- Create a game object that has *MeshRenderer* and *MeshFilter* component, and add your own script component and a *Curve* component to it.
- Note: both *Curve* and *MeshCreator* are part of last week's handout project.
- The mesh creation is simple: we can copy the triangles and uvs from the input mesh. We only need to modify the vertices, and recalculate the normals (and (bi)tangents). So we don't even need to use the *MeshBuilder* class.
- Core code: shown on next slide.

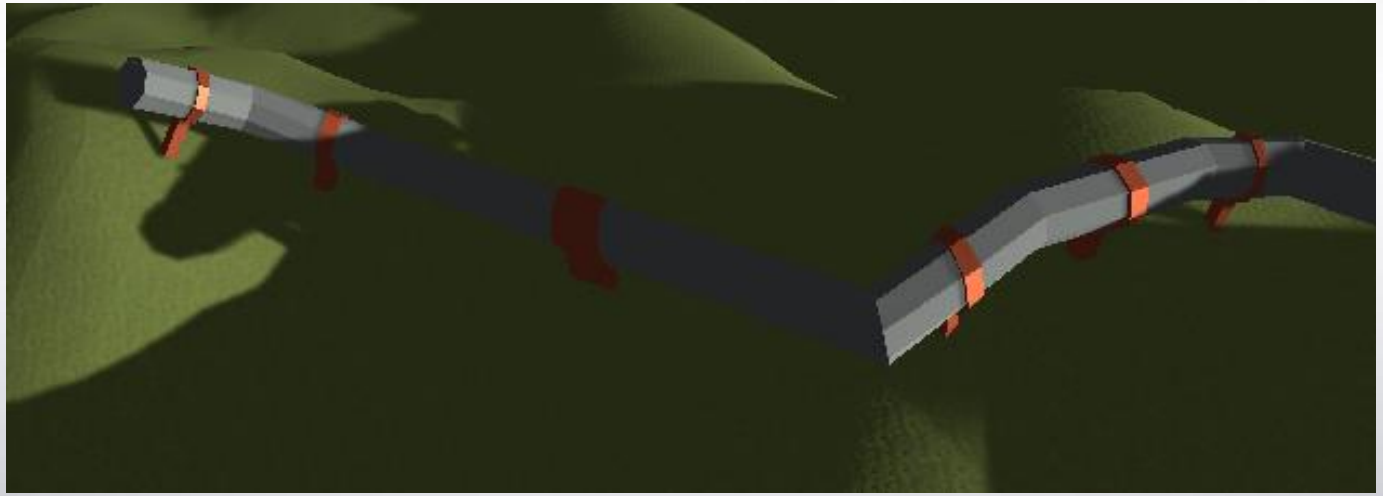
```
public override void RecalculateMesh() {
    Curve spline = GetComponent<Curve>();
    if (spline==null) {
        Debug.Log("WarpMeshQuad: this game object needs to have a curve component");
        return;
    }
    List<Vector3> points = spline.points;
    if (points.Count!=4) {
        Debug.Log("WarpMeshQuad: the curve component needs to have four points");
        return;
    }
    Vector3[] warpedVertices = new Vector3[InputMesh.vertices.Length];

    for (int i = 0; i<InputMesh.vertices.Length; i++) {
        // Do the actual warping (vertex position change - see formula on slides):
        warpedVertices[i] = WarpVertexUsingQuad(InputMesh.vertices[i], points);
    }

    Mesh newMesh = new Mesh();
    newMesh.vertices = warpedVertices;
    newMesh.uv = InputMesh.uv; // Maybe: clone?
    newMesh.subMeshCount = InputMesh.subMeshCount;

    // Just copy the submeshes and triangles:
    for (int i = 0; i<InputMesh.subMeshCount; i++) {
        newMesh.SetTriangles(InputMesh.GetTriangles(i), i);
    }

    newMesh.RecalculateNormals();
    newMesh.RecalculateTangents();

    GetComponent<MeshFilter>().mesh = newMesh;
}
```

# Possible Application



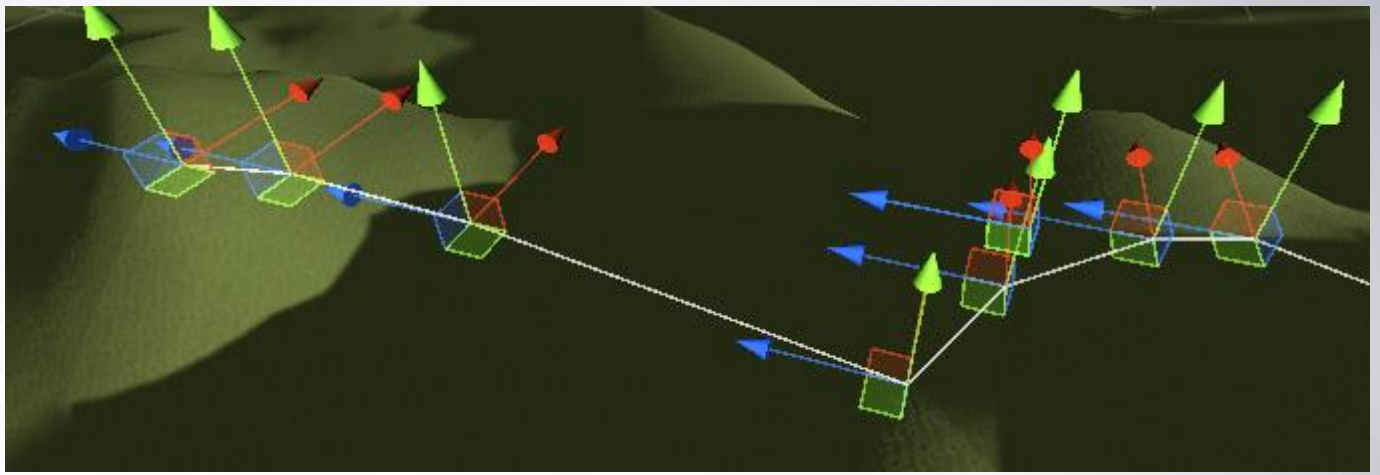
- Note the texture warping and stretching! We'll get to that soon…

# Warping Meshes

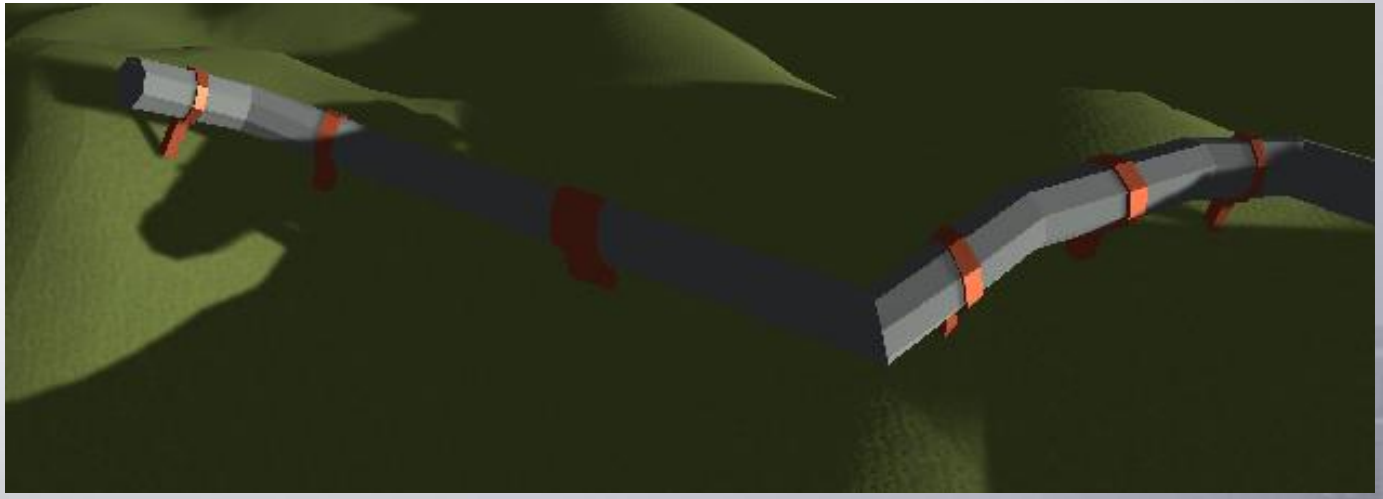


# Warp Mesh along Spline

+



=



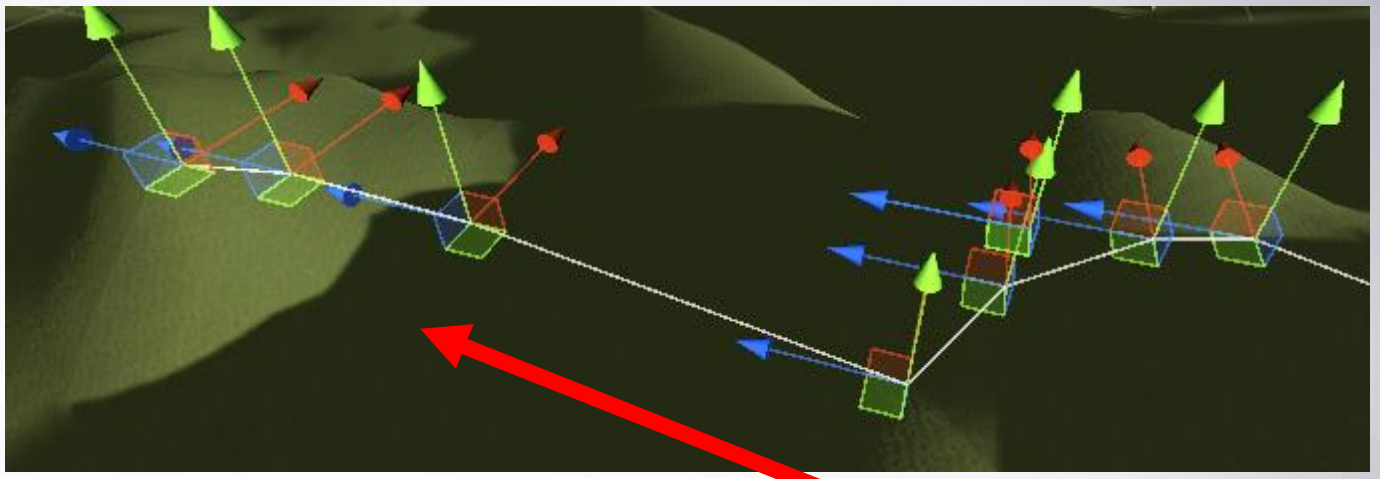
# Step 1: Map Z Coordinate to 0 - 1

InputMesh:

```
Bounds bounds = InputMesh.bounds;
Vector3 max = bounds.max;
Vector3 min = bounds.min;
```

```
for (int j = 0; j<InputMesh.vertexCount; j++) {
    // Map z coordinate to a number t from 0 to 1 (assuming the mesh bounds are correct):
    float t = (InputMesh.vertices[j].z - min.z) / (max.z - min.z);
```

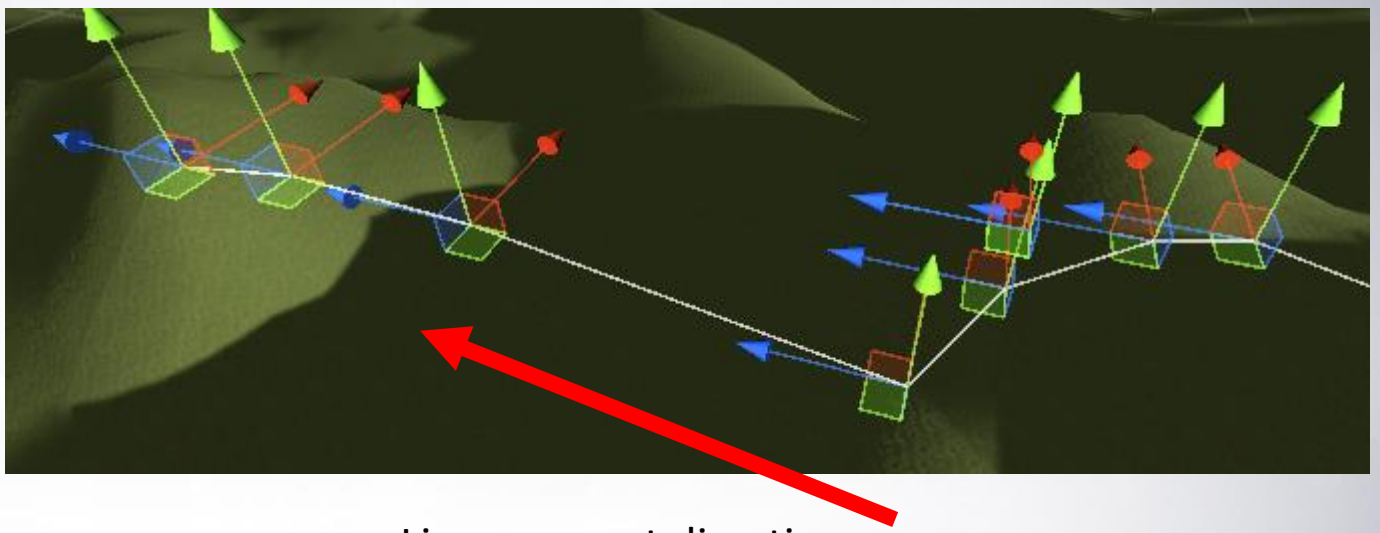
# Step 2: Linear Interpolation between Curve Points




*Question: how is linear interpolation related to affine / convex combinations?*

```
// Use the value t to linearly interpolate between the start and end points of the line segment:
// Choose one of the two lines below - they are completely equivalent!
Vector3 interpolatedLineSegmentPoint = Vector3.Lerp(points[i], points[i+1], t);
Vector3 interpolatedLineSegmentPoint = points[i]*(1-t) + points[i+1] * t; // Lerp = the weighted average between two vectors
```

# Step 3: Get the Orientation of a Curve Segment




```
var localOrientation = new List<Quaternion>();
for (int i = 0; i < points.Count-1; i++) {
    // Compute a unit length vector from the current point to the next:
    Vector3 lineSegmentDirection = (points[i+1]-points[i]).normalized;
    // Store a matching orientation (computing an orientation requires a forward direction vector and an up direction vector):
    localOrientation.Add(Quaternion.LookRotation(lineSegmentDirection,Vector3.up));
}
```

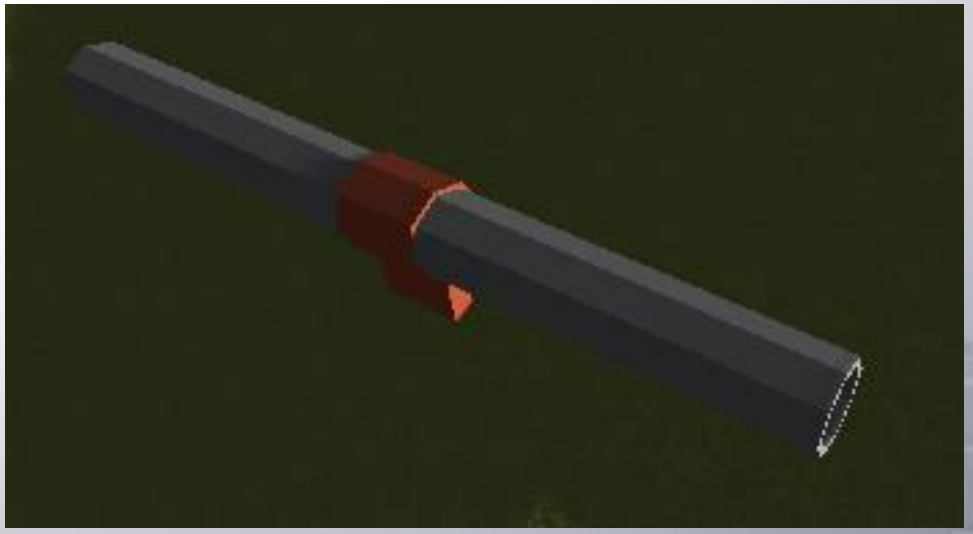
# Step 4: Rotate the XY coordinates of the Model

```
// Center and scale the input mesh vertices, using the values given in the inspector:
Vector3 inputV = (InputMesh.vertices[j] - MeshOrigin) * MeshScale;
// Set the z-coordinate to zero:
inputV.Scale(new Vector3(1, 1, 0));
```

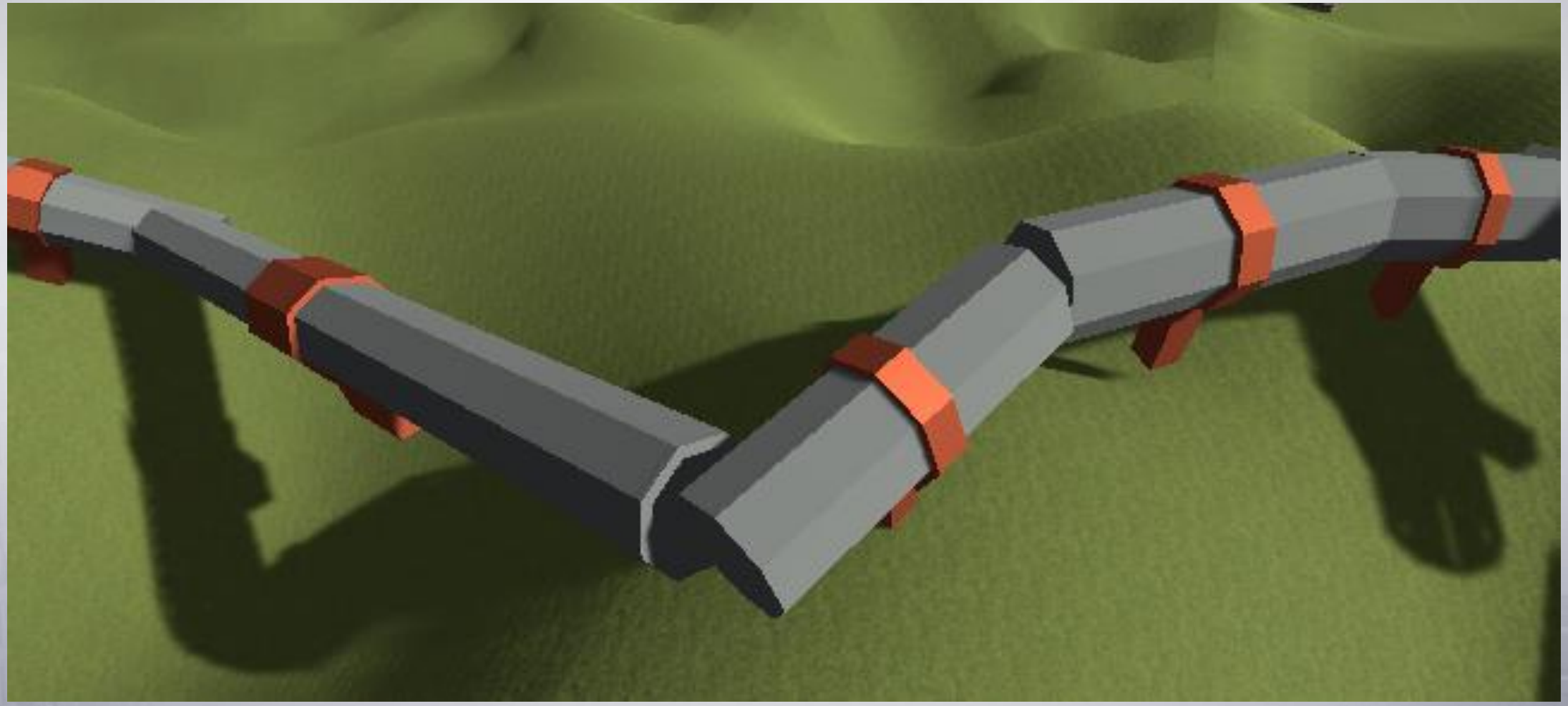
```
Vector3 rotatedXYModelCoordinate = localOrientation[i] * inputV;
```

Recall from last week: this is how you rotate a point in Unity: multiply a quaternion with a vector!

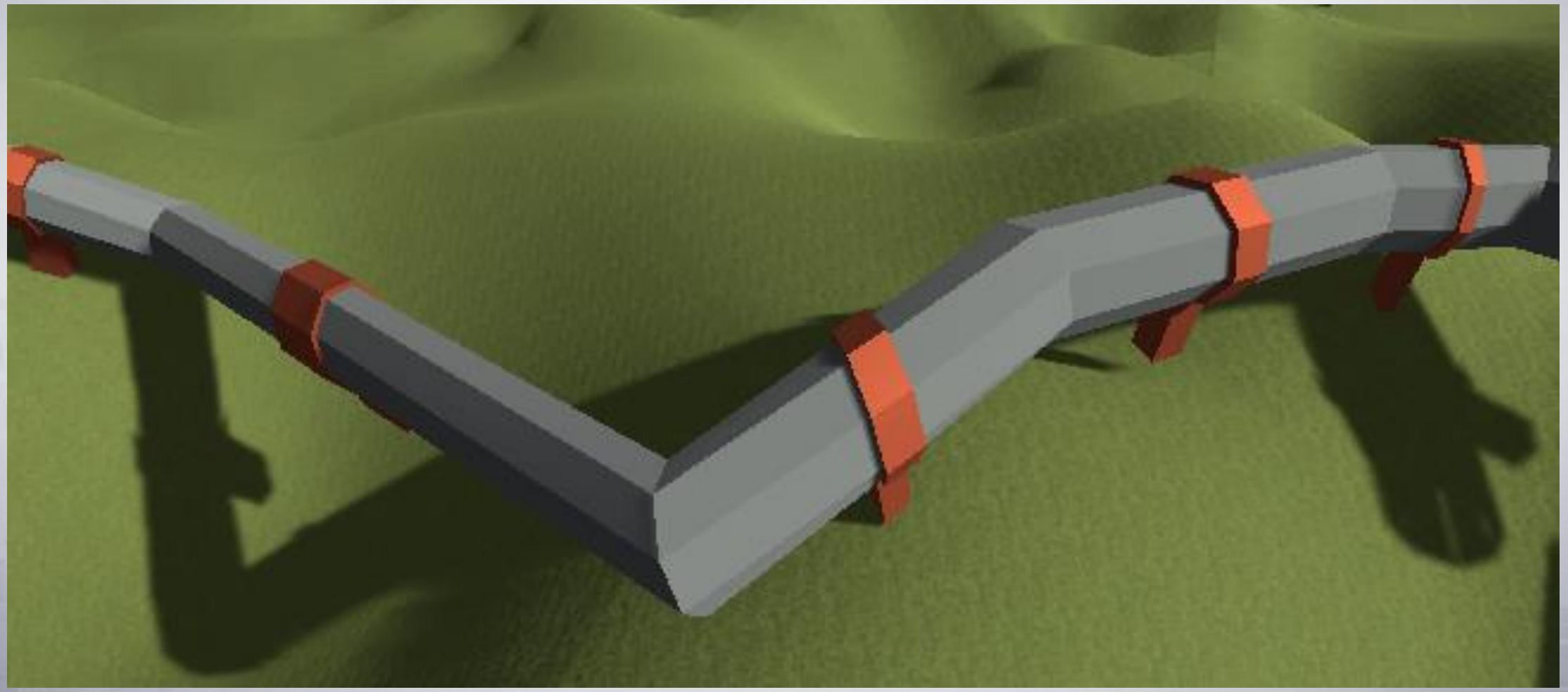
Adding the *rotatedXYModelCoordinate* and *InterpolatedSplinePoint* together gives a stretched + rotated pipe piece:



# Result



# Desired Result



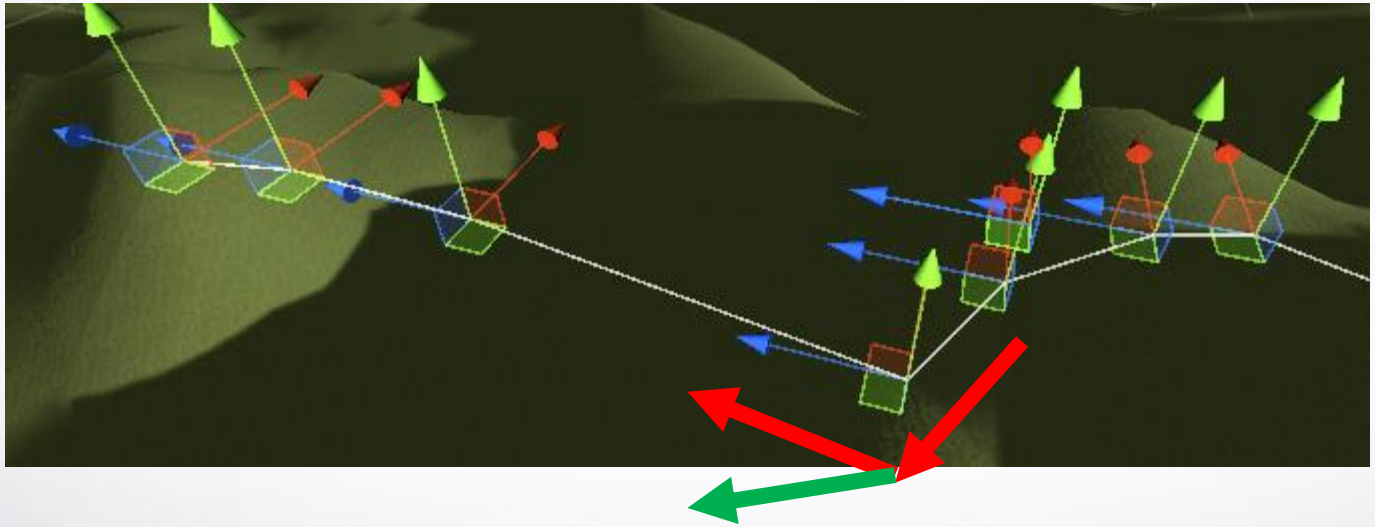
# Solution

- Just like we can apply linear interpolation to points (vectors), we can also apply it to orientations (quaternions)!
- See https://docs.unity3d.com/ScriptReference/Quaternion.Lerp.html

Tips:

- For each curve *point,* compute a direction vector which is the *average* of the direction vectors of the two incident line segments (see next slide)
- Use those direction vectors to compute orientations (quaternions) for each *point*
- Linearly interpolate between those orientations, with Quaternion.Lerp, using *t*

# Point Directions



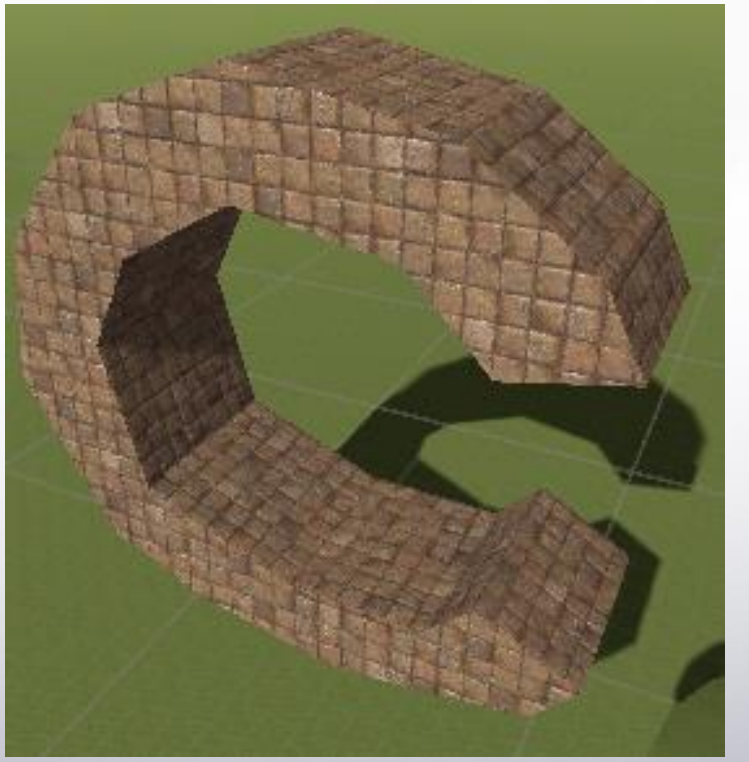
Red = Line segment directions (normalized): d1 and d2

Green = average: ((d1+d2)/2).normalized

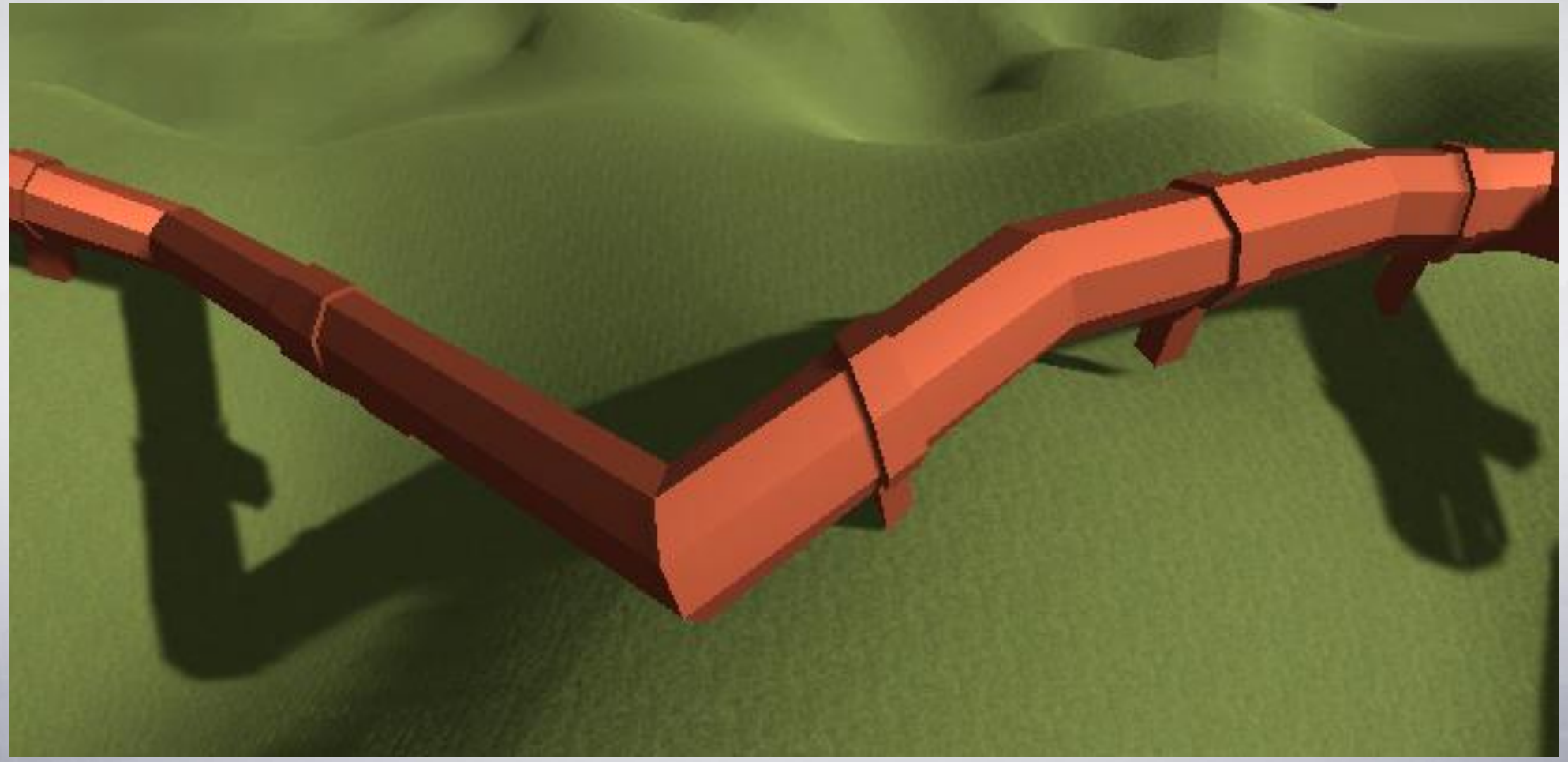
# Possible Applications

- By warping meshes along a curve, you can make e.g.:
  - Road systems (see the *HorizontalCurves* scene in the handout!)
  - City walls
- You can build your city on a terrain this way!
- However, we still have some trouble with texture stretching and warping → next topic

# Texturing Procedural Meshes

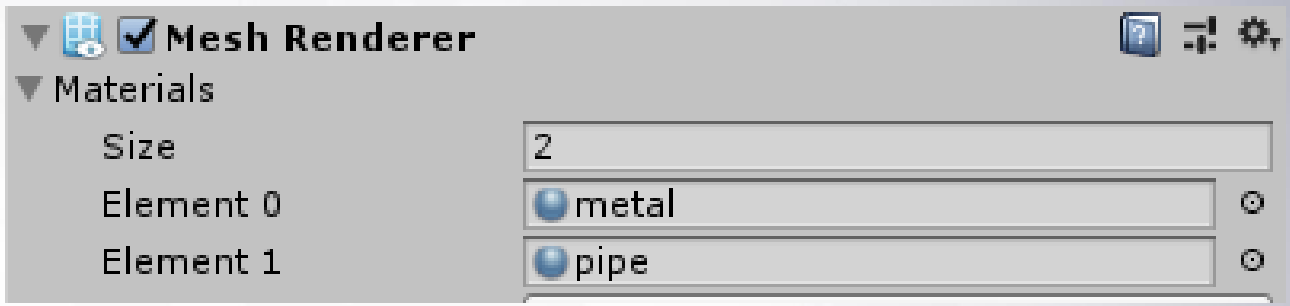


# Problem – Only One Material Used



# Using Multiple Materials: Submeshes

- Unity's Mesh Renderer can use multiple materials
- A mesh can consist of *submeshes*
- Mesh *x* is rendered using material *x*

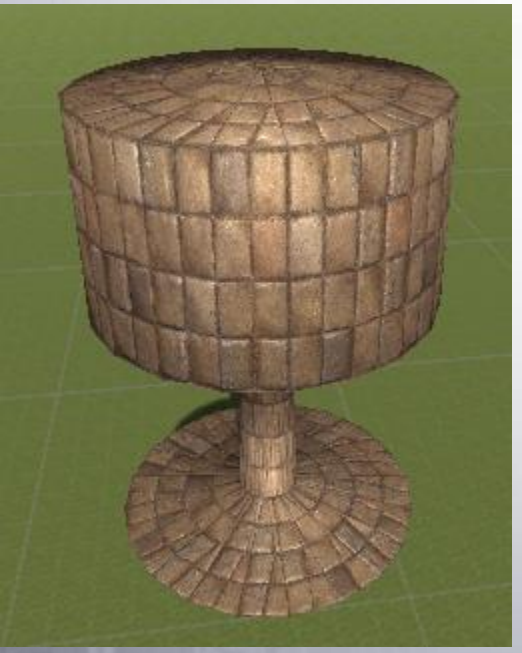


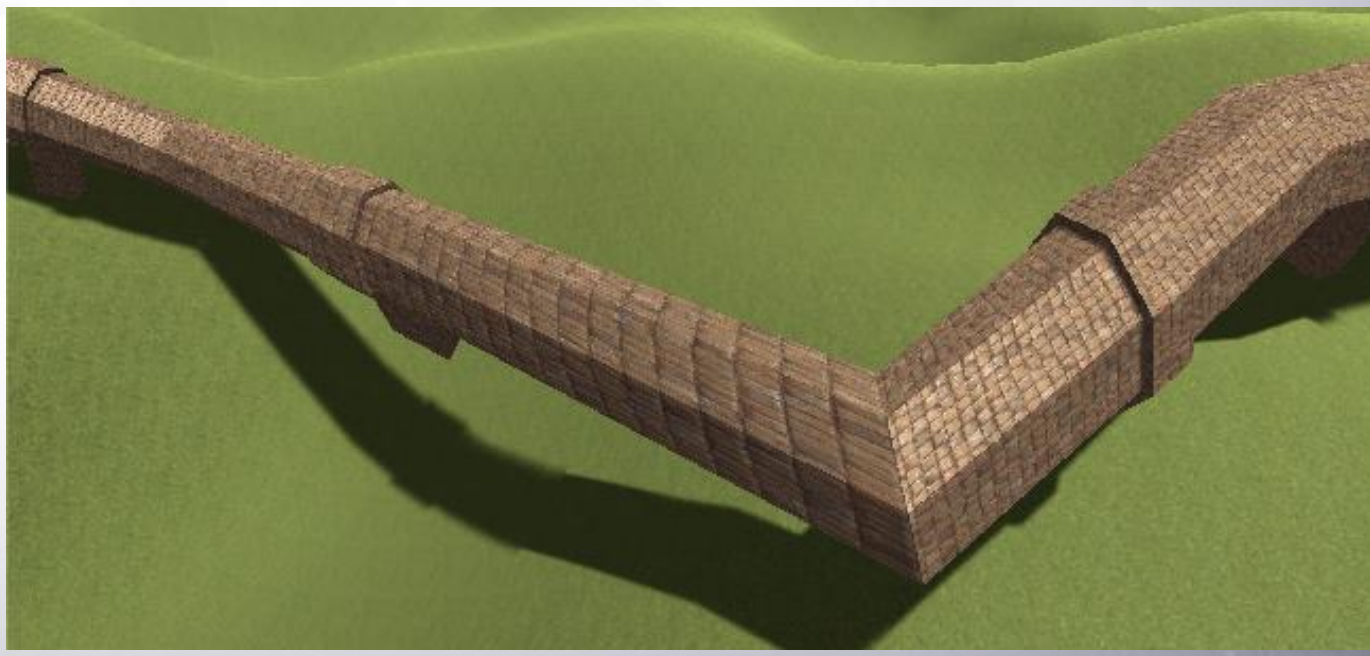

- Use **Mesh.GetTriangles(int submesh)** to get the array of triangles (vertex indices) that belong to a certain submesh
- Use **Mesh.SetTriangles** to assign a triangle array to a certain submesh
- MeshBuilder already supports submeshes: see the **AddTriangle** method

- More information:
  - https://docs.unity3d.com/ScriptReference/Mesh.GetTriangles.html
  - https://docs.unity3d.com/ScriptReference/Mesh.SetTriangles.html

# Texture Coordinates / UVs

- One of the most challenging aspects of procedural meshes is getting the uvs correct!
- Possible basic approaches:
  - Choose uvs in a logical way, considering the method (e.g. lathe, extrude)
  - Copy uvs from input mesh

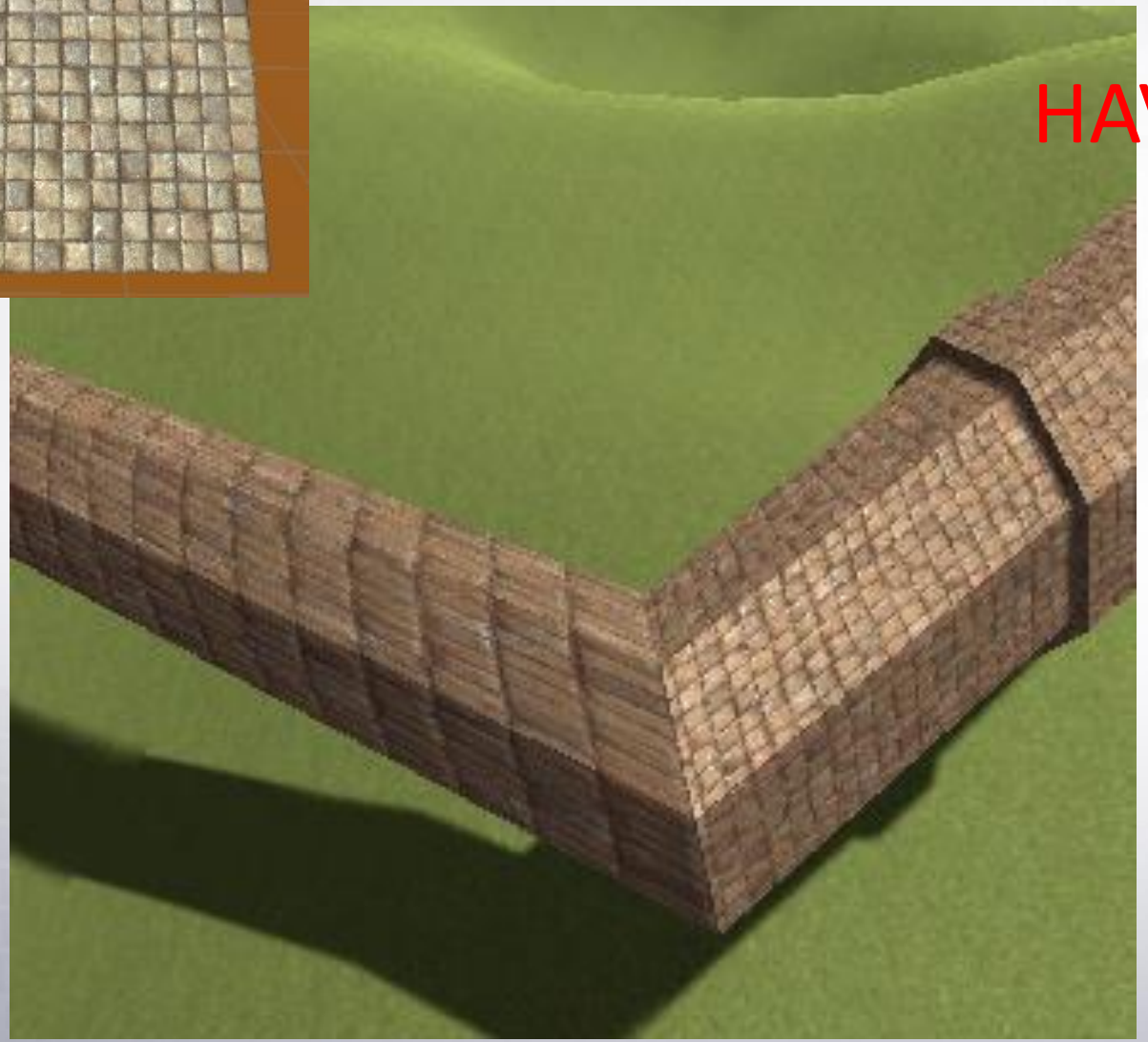




# Problem: Texture Stretching

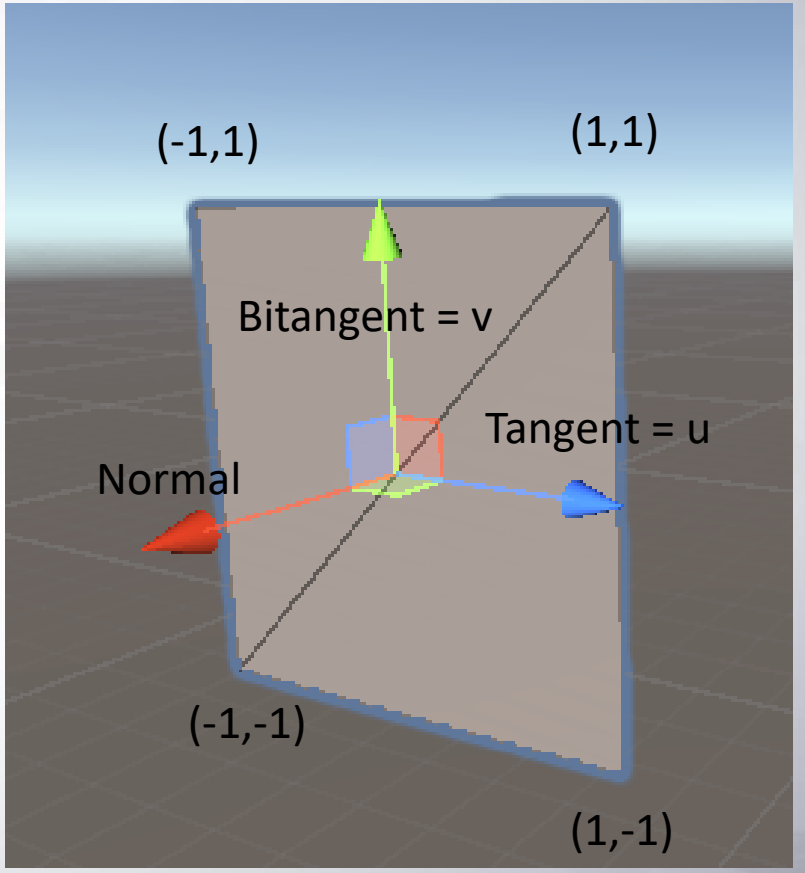
WANT

HAVE

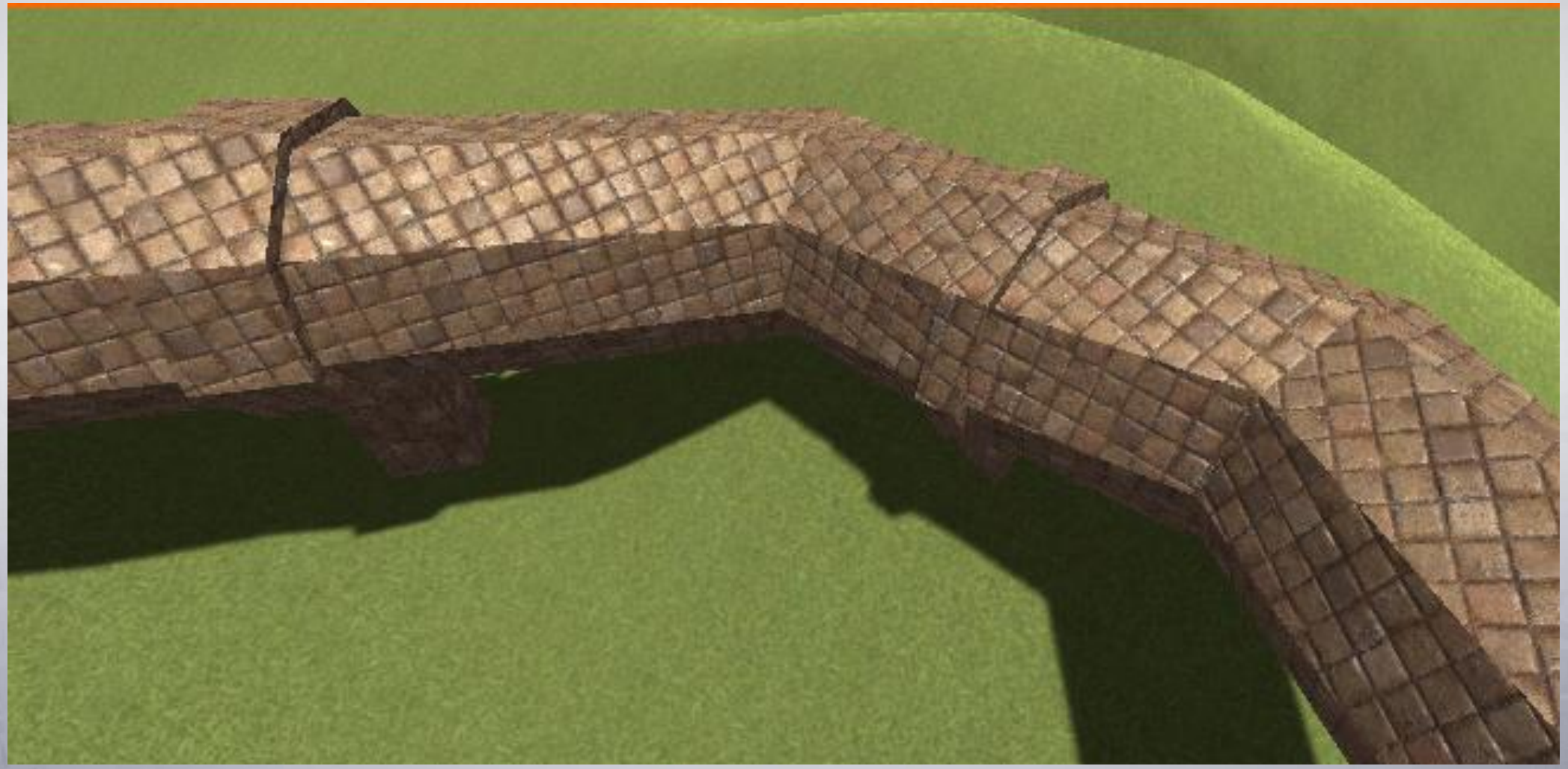


# Possible Approach

- For each triangle, compute:
  - The *normal* vector
  - Two vectors that lie in the same plane as the triangle:
    - *Tangent*
    - *Bitangent* (also called *binormal)*
  - All these vectors are perpendicular to each other!
  - Recall from last week: you can use the *cross product* + *normalize* to get a vector that is perpendicular to two other vectors
- For each point of the triangle, the uvs are the coordinates relative to the *tangent* and *bitangent* (→ vector projection)
- (Details and proofs: 3D Math)
- Implementation: *AutoUv.cs*

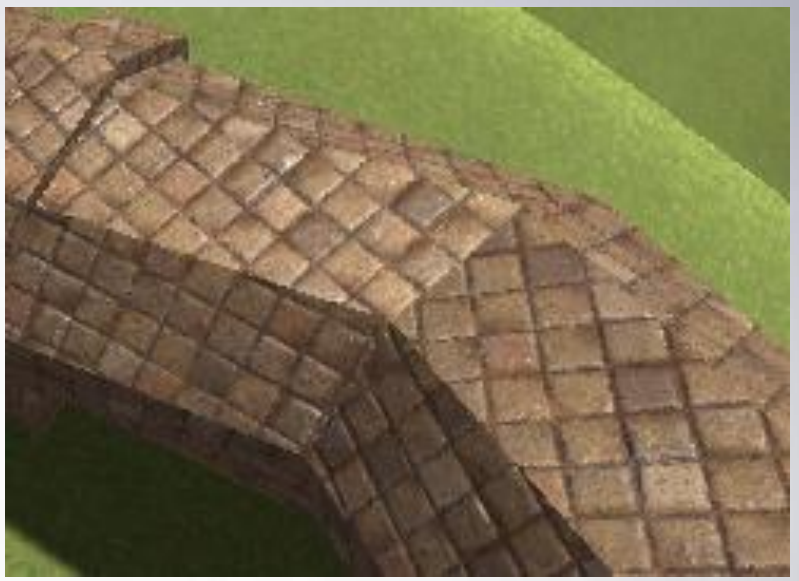


# Result (Demo)

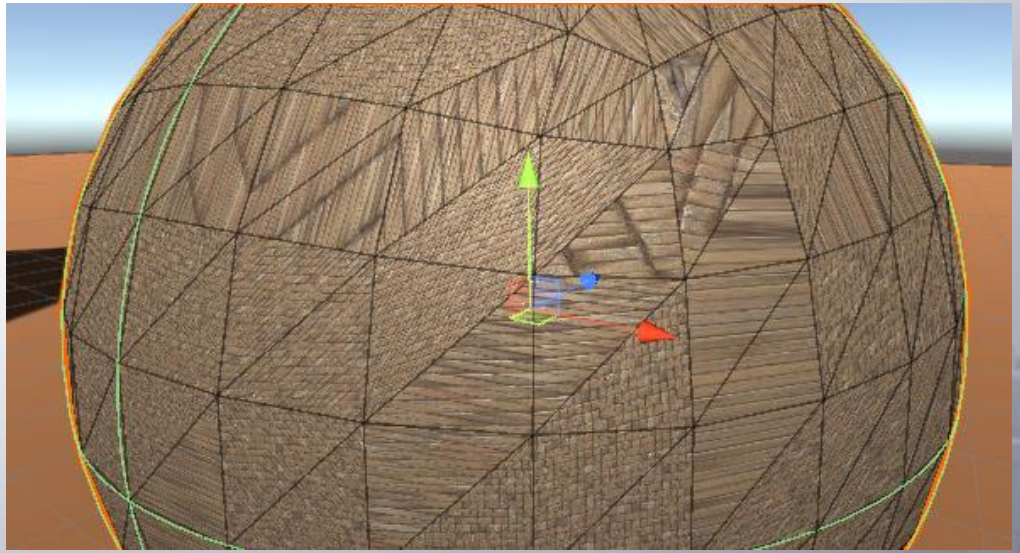


# Problems

- The transitions between triangles are random



- This doesn’t work at all when triangles with different normals share vertices!

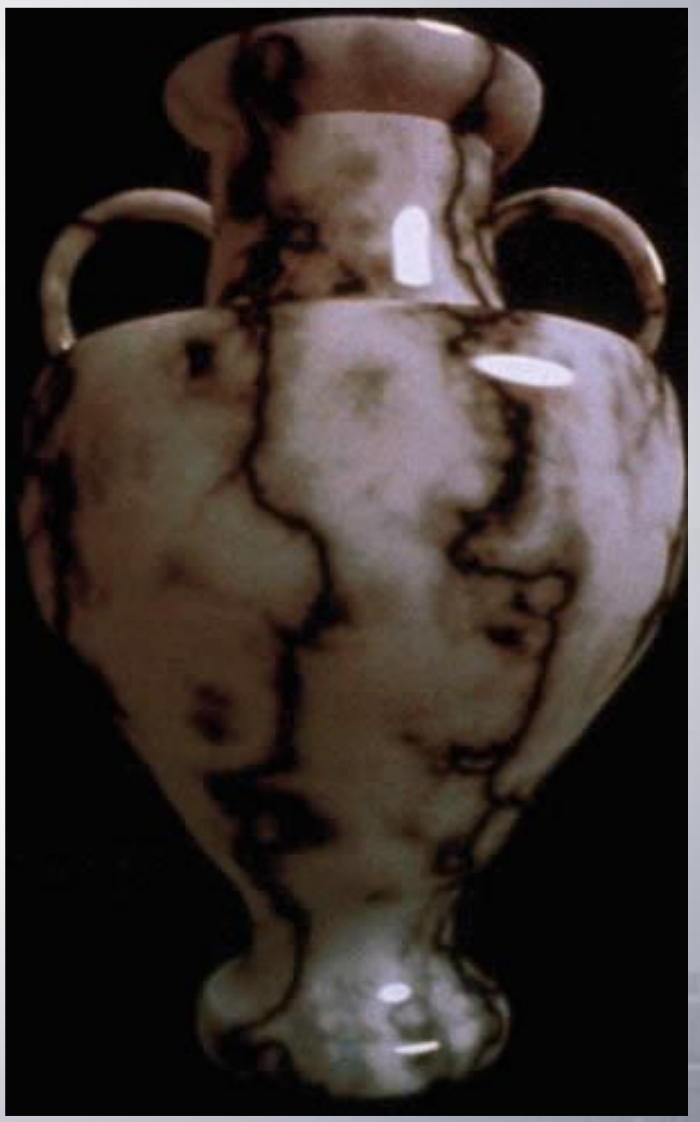


- Unfortunately, there is no magic one-size-fits-all solution for uv-ing & texturing meshes…
- …maybe we should think outside the box?

→ Use *shader based* solutions, instead of textures!

# Solid Objects

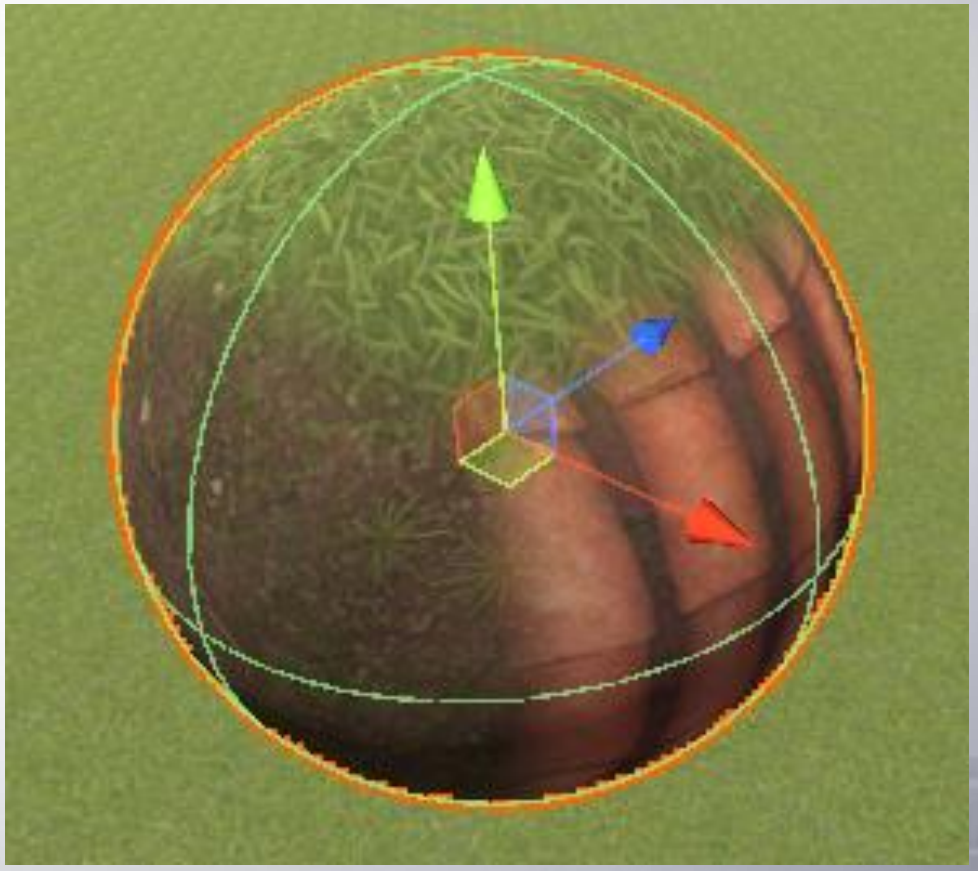
- A possible approach is to use the fact that real life objects are made from 3D materials, which have their own distinctive 3D patterns. Examples:
  - wood rings
  - Marble veins
- Method: In a shader, use the *3D vertex positions* to compute a color (e.g. using a 3D texture, or a 3D noise function)



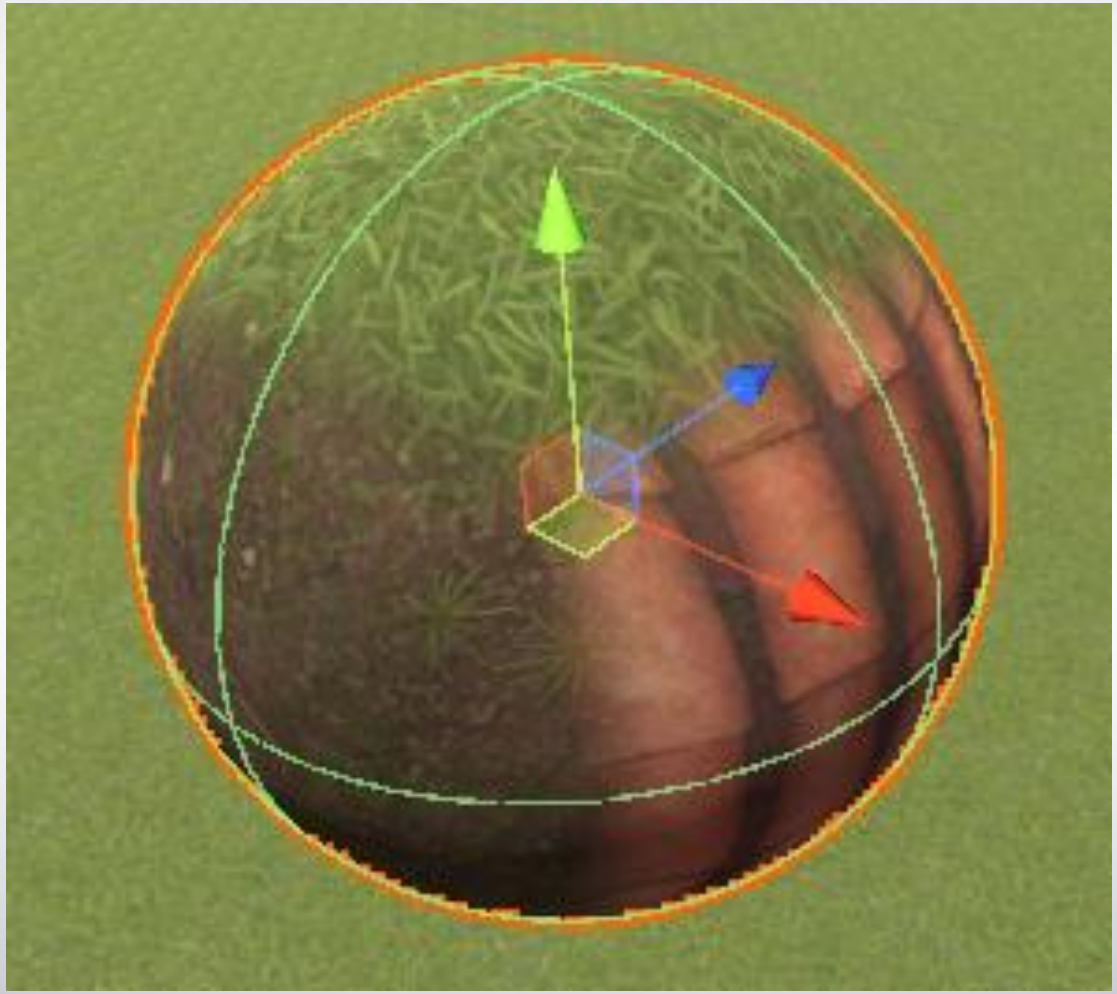
# TriPlanar Texturing

- Approach: in a shader, use the *world normal* to blend between textures
  - Mainly in x-direction:
    - Use y&z as texture coordinates
    - Use texture 1
  - Mainly in y-direction:
    - Use x&z as texture coordinates
    - Use texture 2
  - Mainly in z-direction:
    - Use x&y as texture coordinates
    - Use texture 3

- A surface shader that implements this idea is part of the handout!



# Demo



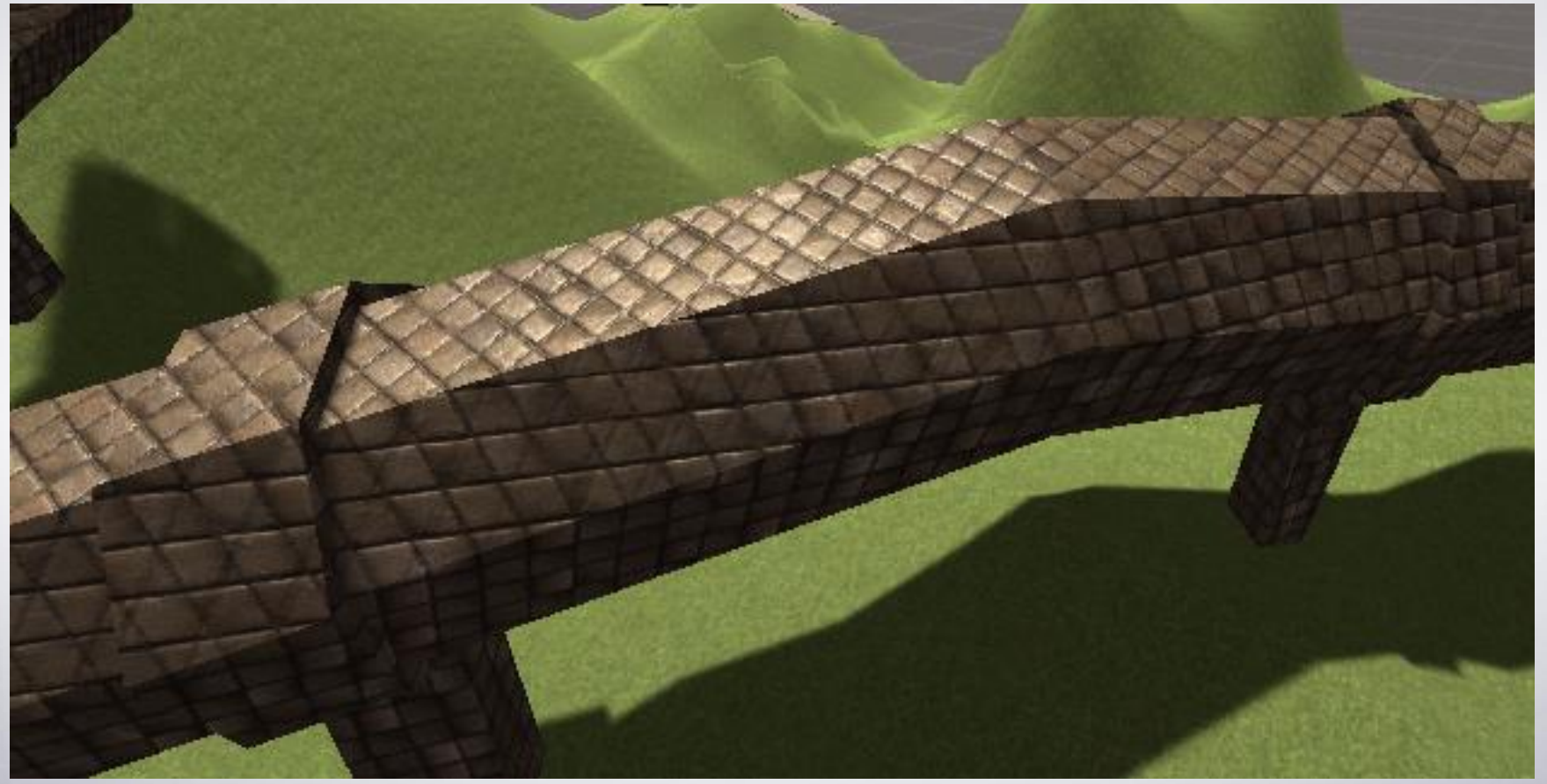
# (Surface) Shader Code

```
// Use absolute values of the world normal vector as start point for the blend factors:
float3 blend = abs(IN.worldNormal);

// Higher power leads to sharper texture boundaries & less blending:
blend = pow(blend, _Sharpness);

// For a correct blend between textures, set the sum of blend factors to 1:
blend /= (blend.x + blend.y + blend.z);

// calculate triplanar uvs:
float2 uvX = IN.worldPos.zy / _TextureScale;
float2 uvY = IN.worldPos.xz / _TextureScale;
float2 uvZ = IN.worldPos.xy / _TextureScale;

// Simple albedo textures:
fixed4 colX = tex2D(_MainTexX, uvX);
fixed4 colY = tex2D(_MainTexY, uvY);
fixed4 colZ = tex2D(_MainTexZ, uvZ);

// Final color: a blend between the three textures:
fixed4 col = colX * blend.x + colY * blend.y + colZ * blend.z;
```

If you understand the idea, you can implement this in *shader graph* as well!

Actually, shader graph spoils the fun by already having a triplanar node…

# Result



More information / advanced triplanar shader:
https://medium.com/@bgolus/normal-mapping-for-a-triplanar-shader-10bf39dca05a

# Assets and Scenes

# Leaking Meshes

- Using the given editor scripts (CurveEditor.cs), you can create meshes in edit mode
- When re-opening the project, the meshes are still there
- …but they are not part of the assets!
- Where are they stored?

→ *The created meshes are stored in the scene!*

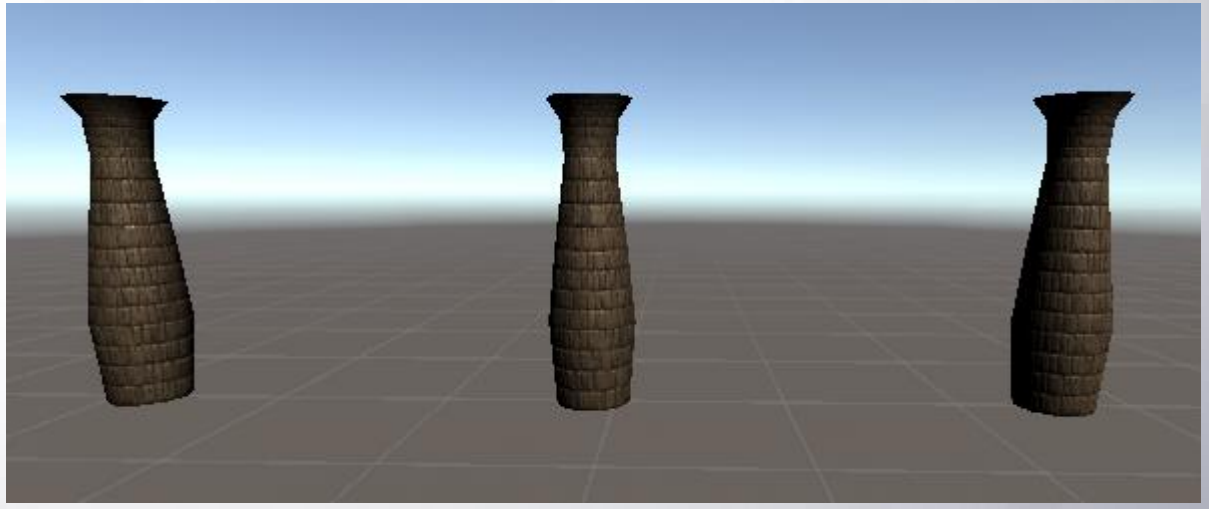
- Open the scene file in a text editor to see how it's done…
- That's why the scene files are huge!

- This is not the ideal way of doing it
- Unity may complain about "leaking meshes into the scene"

# Possible Solutions

1. Modify the “SharedMesh”
   - This modifies all game objects that use this mesh!
   - Can be dangerous!
   - …but also useful… (saving memory, simultaneous editing)

   - Demo



2. Export the created mesh to an asset
   - Unity method: AssetDatabase.CreateAsset
   - Independent of Unity: create OBJ file

# Save OBJ

- The *OBJ format* is a human readable format that is supported by many 3D programs, including Unity.
- The *SaveMeshToObj* script saves the (generated) mesh to an OBJ file
- You can open the OBJ file in a text editor: this shows the
  - Vertex positions *(v)*
  - UVs *(vt)*
  - Vertex normals *(vn)*
  - Triangles / polygons *(f)*

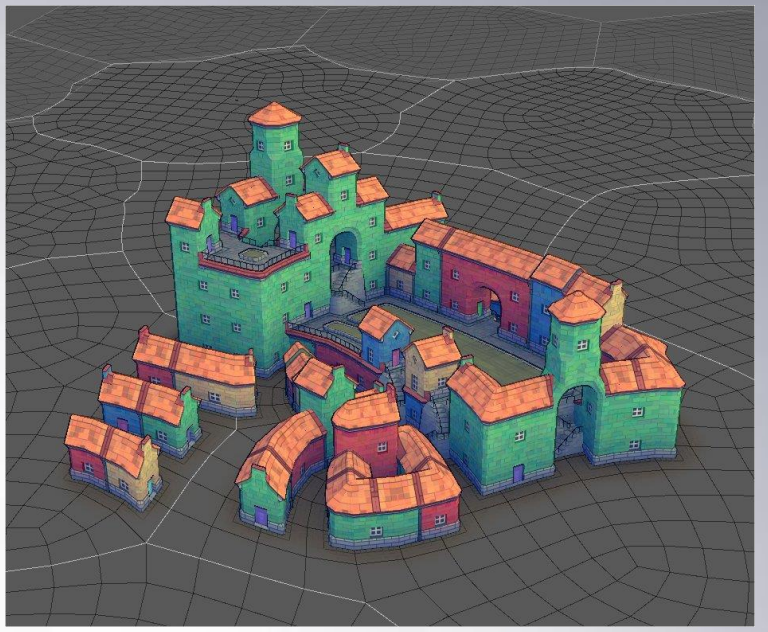
- See the *CombineMeshes* scene in the handout project!

# Conclusion

# Summary

- Last week: introduction to mesh creation in Unity. Vertices, normals, UVs, triangles.
- Today we saw more advanced methods of creating / modifying meshes procedurally:
  - Warp Mesh on Quad / Warp Mesh along Curve
  - Methods that modify existing meshes are especially useful: combining human input with procedural methods
- Texturing procedural meshes is always challenging: texture stretching, warping, stitches… Possible solutions:
  - Auto UV assignment: UV based on normal / (bi)tangent vectors
  - Shader based solutions: 3d coordinates / triplanar texturing
- How can procedural meshes be stored? Storing in the scene is not ideal…
  - Sharing (procedural) meshes between game objects
  - Create assets: Unity methods such as CreateAsset, or save as OBJ files
- *Starting point for nearly all of these methods are in last week's handout!* (see also the PDF)
- *Nearly all procedural methods that are not grid-based require some (vector) math understanding*

# What now?



- Finish the scripts! Look at
  - the //TODO items in the code
  - the information on these slides
  - the PDF explaining the project!

- We focused on editor tools, but *curves* can easily be generated at runtime as well! (Be creative!)

- Possible applications for city building:
  - Odd shaped buildings (polygon shapes)
  - Creating road systems that aren't rectangular

- For upcoming projects:
  - Create + use Unity editor tooling to quickly create scenes!
  - *Bonus tip:* The *WarpMeshAlongSpline* script is an excellent tool for creating tracks for a race game!!!