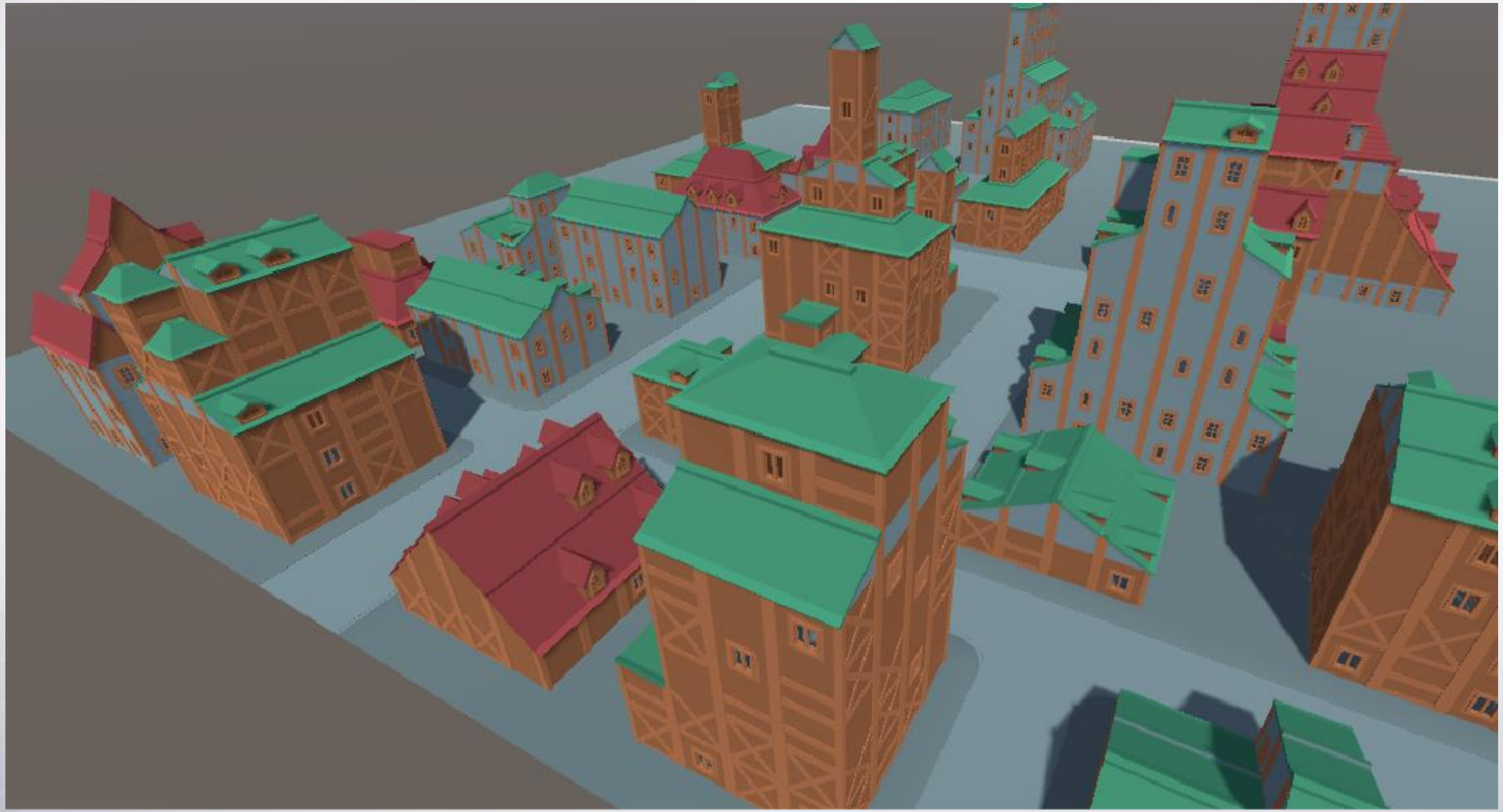
PROCEDURAL ART

(Scripting-oriented) LECTURE 4 – Mesh Creation Basics

by Paul Bonsma

March 28, 2022

Recap Week 2: Modular Meshes



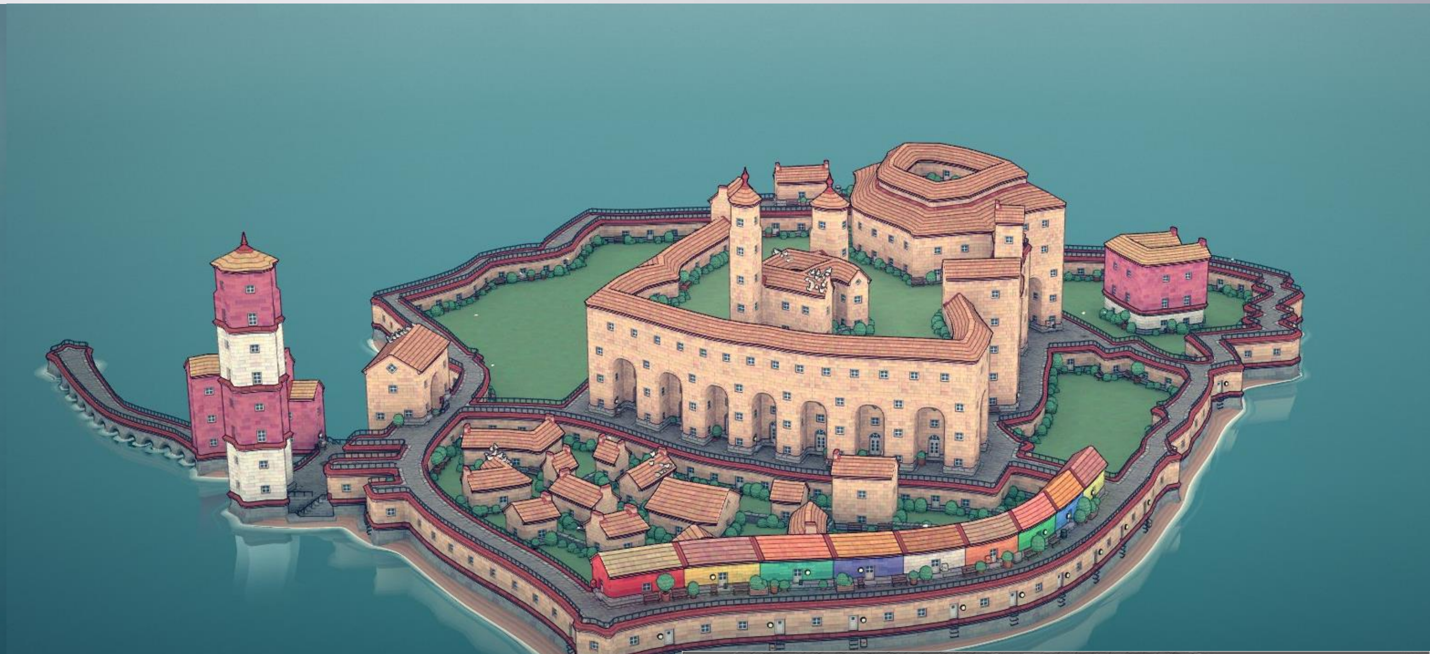
Challenge

- Problem: when using modular meshes, shapes tend to be very “rectangular” (even though you can scale & rotate the building blocks)
  - Rectangular buildings
  - Rectangular roads

- We would like to get shapes as shown on the right!
  - Polygon shaped buildings
  - Curved roads

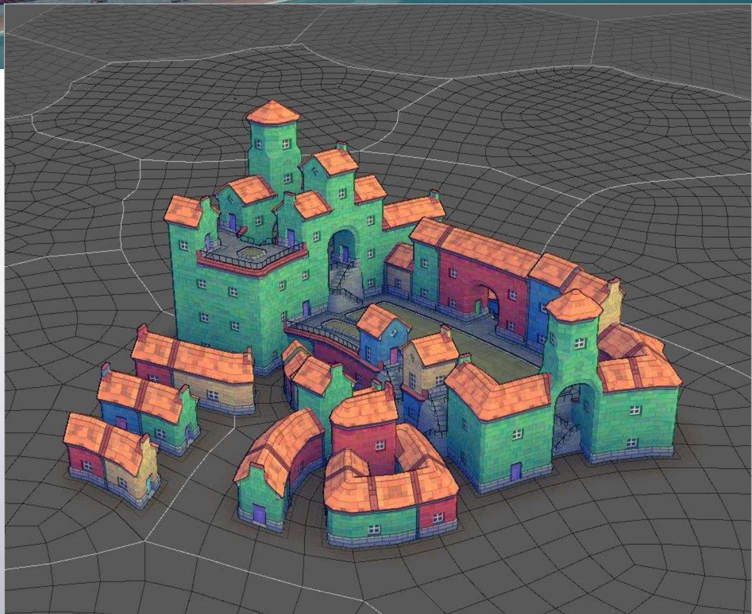
- This requires *procedural generation or modification of meshes*

Townscaper by Oskar Stålberg (June 2020, Steam)



https://www.youtube.com/watch?v=kI1GkS0l2kY
https://twitter.com/osksta/status/1184721532712554496
https://twitter.com/OskSta/status/1151152046055272449
https://store.steampowered.com/app/1291340/Townscaper/

Note: Get inspired, maybe buy it on Steam and have fun with it, but *don't try to make something like this yourself!* (Except maybe for your minor)



More Procedural Meshes

Spore (2008)

No Man’s Sky (2016)

Outline (Next two lectures)

- Mesh basics: vertices, triangles
- uvs and normals
- Lathe & curves
- Extrude & triangulation

- Warping meshes
- Texturing procedural meshes
- Assets and scenes

“How does it help me pass the course?”

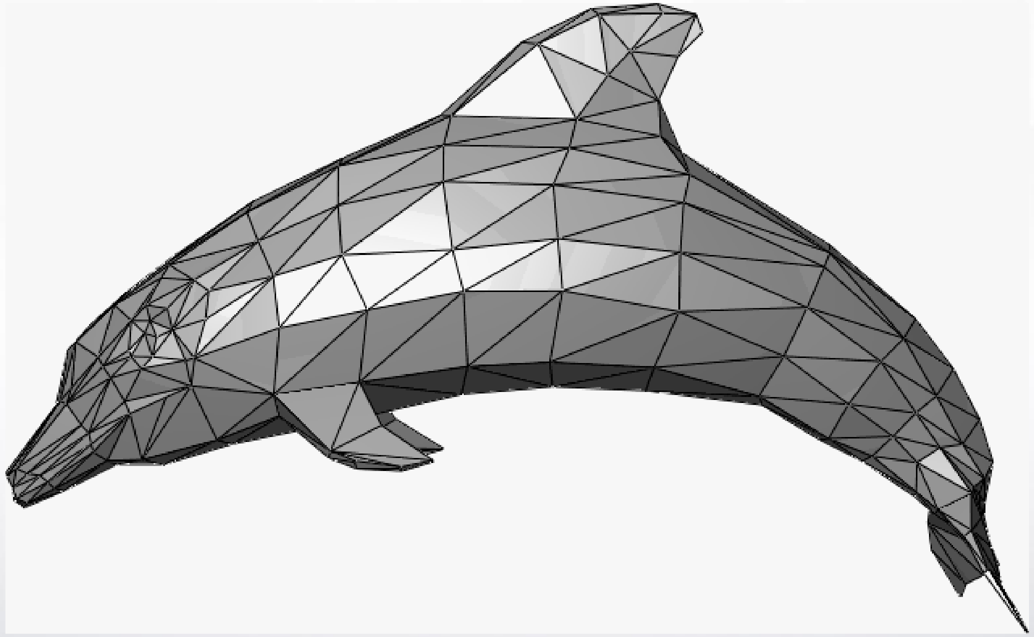
These two lectures + handout will help you directly towards these grading criteria:

- Meshes are created or modified procedurally (e.g. lathe, extrude, warp).
- Procedural meshes are textured without extreme artefacts, e.g. stretching, stitches (this can be done with procedural UVs or by shaders).
- All textures have the proper scale (as applied in the scene).
- There is custom (Unity editor) tooling for fast scene creation (e.g. building placement, road drawing).

They might help you to achieve these criteria:

- The resulting structures match the visual research.
- Optimizations have been done for real-time efficiency (mesh welding).
- A wide range of shapes is created procedurally from a smaller range of building blocks.

3D Meshes

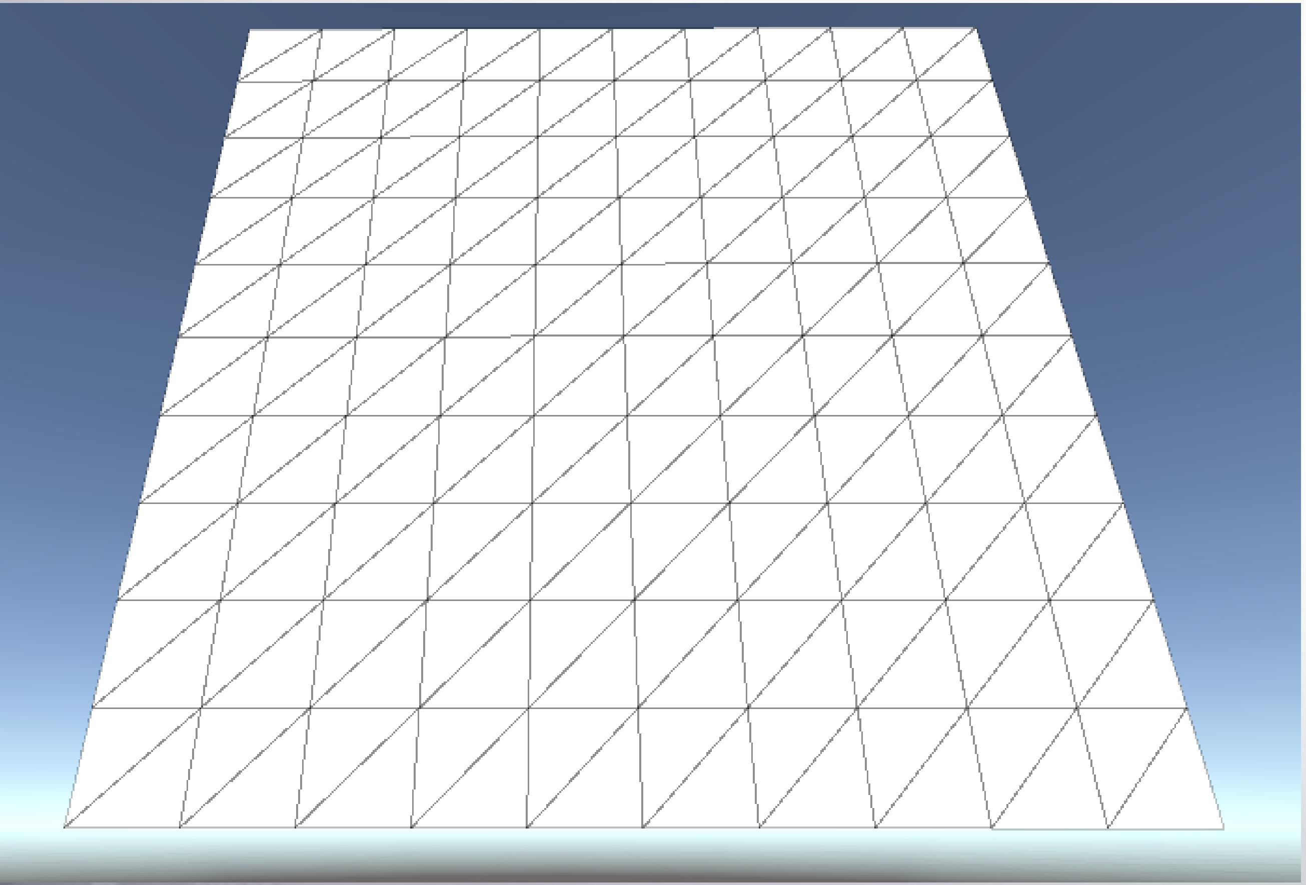


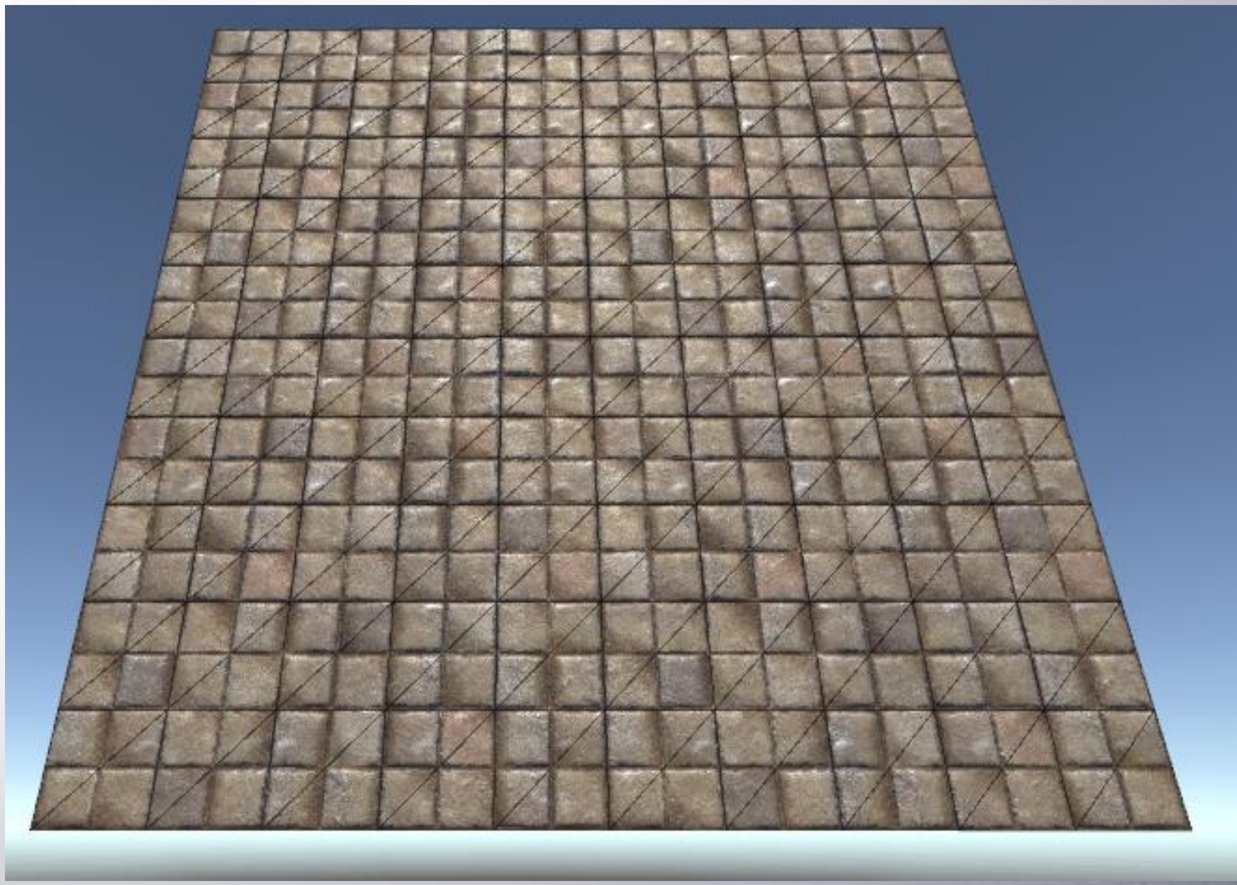
Knowledge Test

- This course was moved to quarter 3, so it’s not preparation for 3D Rendering anymore… (Which made 3D Rendering a bit harder, but this course a bit easier)
- Let’s see how much you still know about meshes…

Quiz: Good Mesh Example

- We wanted to create a basic plane, with 200 triangles and 100 vertices, and a tile material, but on the next slides, something went wrong… What?



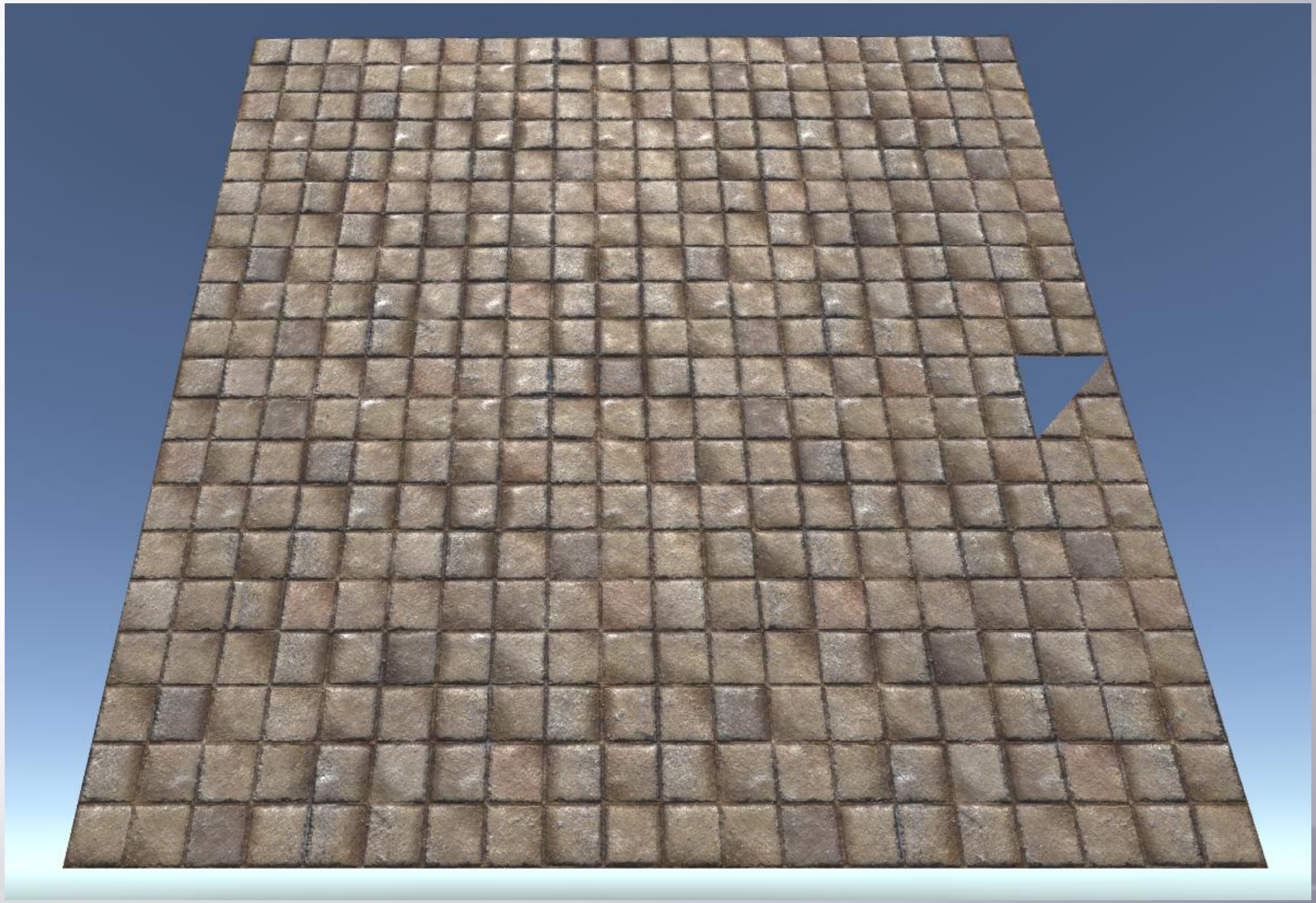


What’s Wrong?

1. There’s a bad UV
2. There’s a bad normal
3. There’s a bad vertex (position)
4. There’s a bad triangle (vertex index)
5. There’s a bad triangle (winding order)

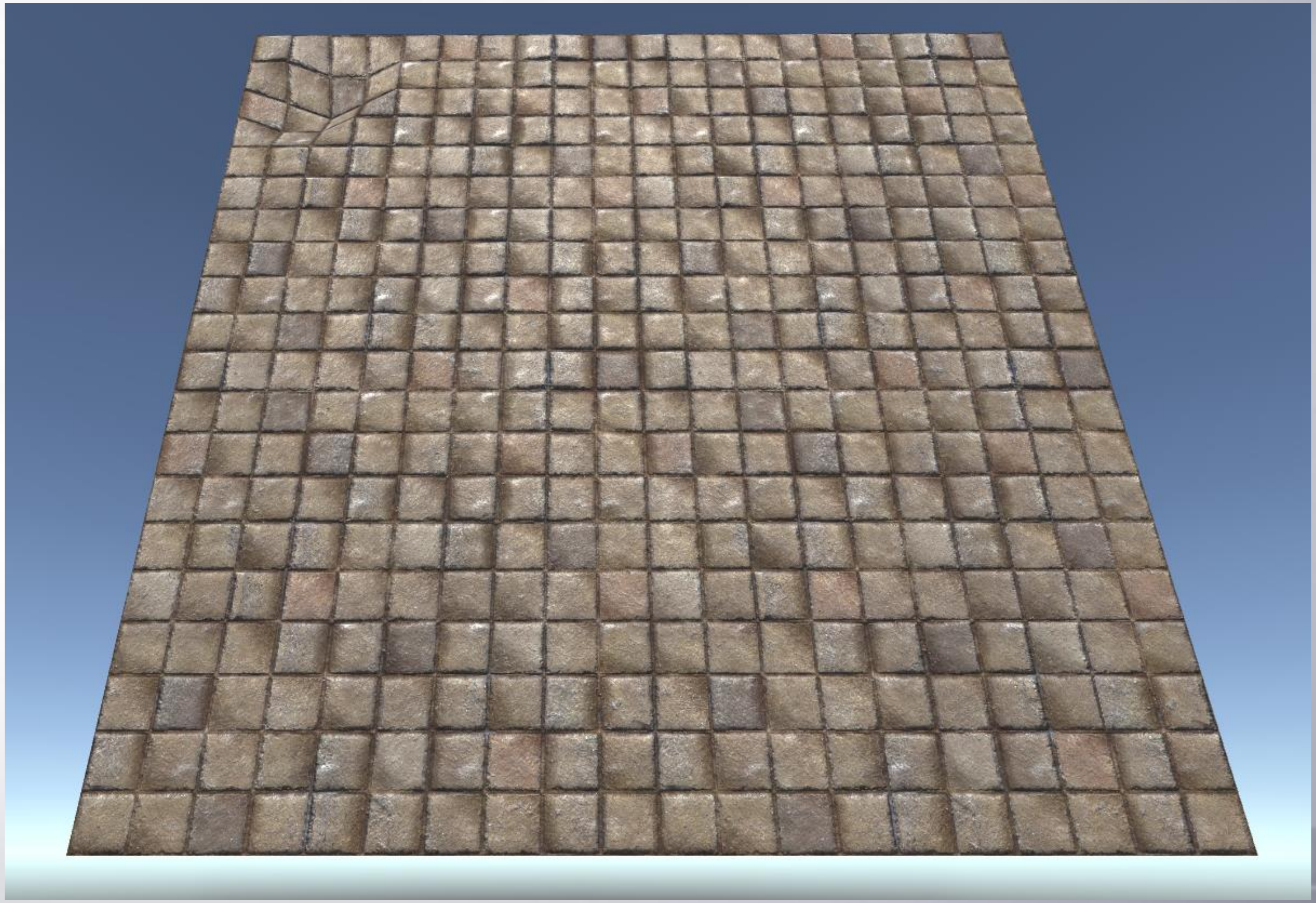
What’s Wrong?

1. There’s a bad UV
2. There’s a bad normal
3. There’s a bad vertex (position)
4. There’s a bad triangle (vertex index)
5. There’s a bad triangle (winding order)



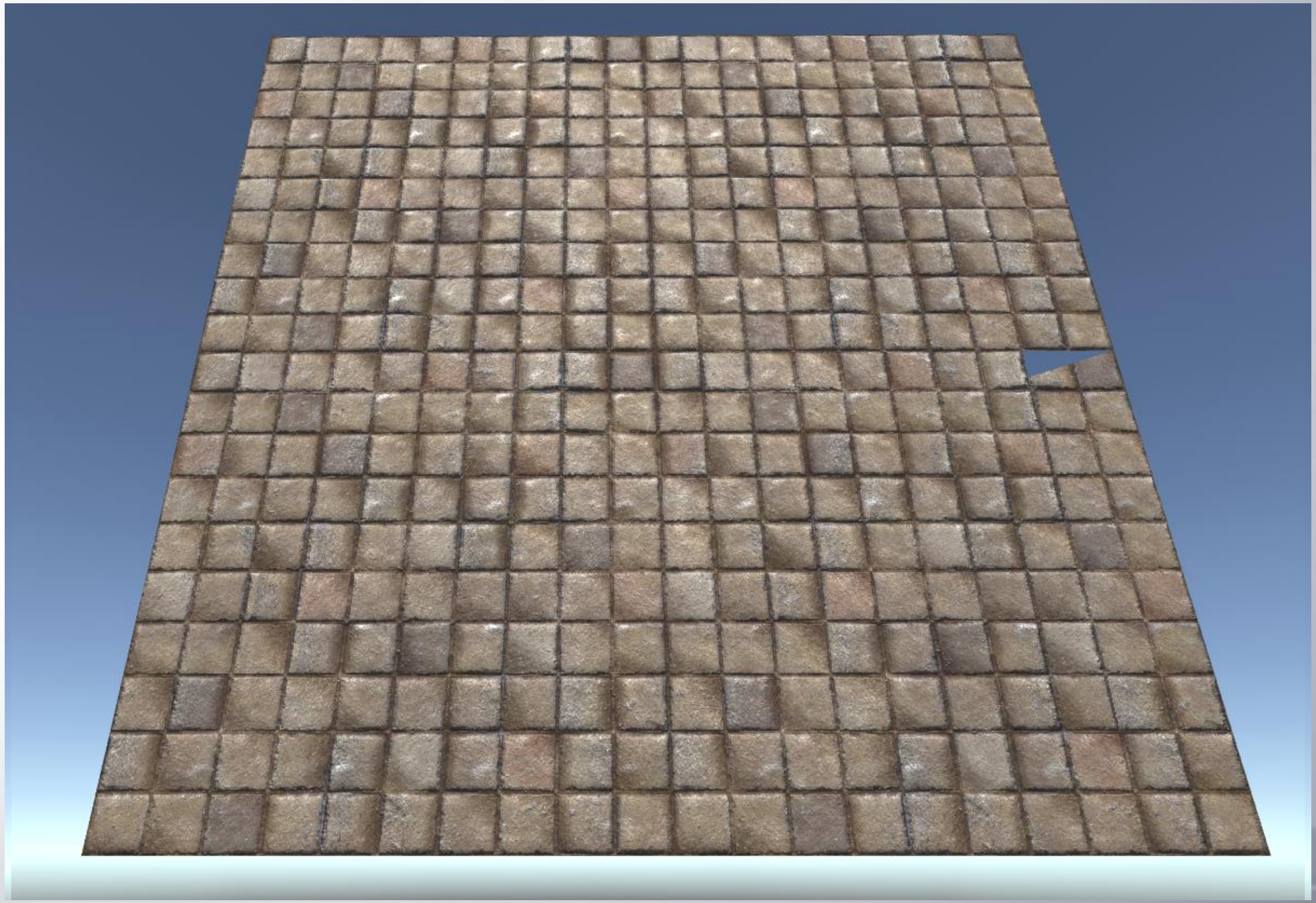
What’s Wrong?

1. There’s a bad UV
2. There’s a bad normal
3. There’s a bad vertex (position)
4. There’s a bad triangle (vertex index)
5. There’s a bad triangle (winding order)



What’s Wrong?

1. There’s a bad UV
2. There’s a bad normal
3. There’s a bad vertex (position)
4. There’s a bad triangle (vertex index)
5. There’s a bad triangle (winding order)

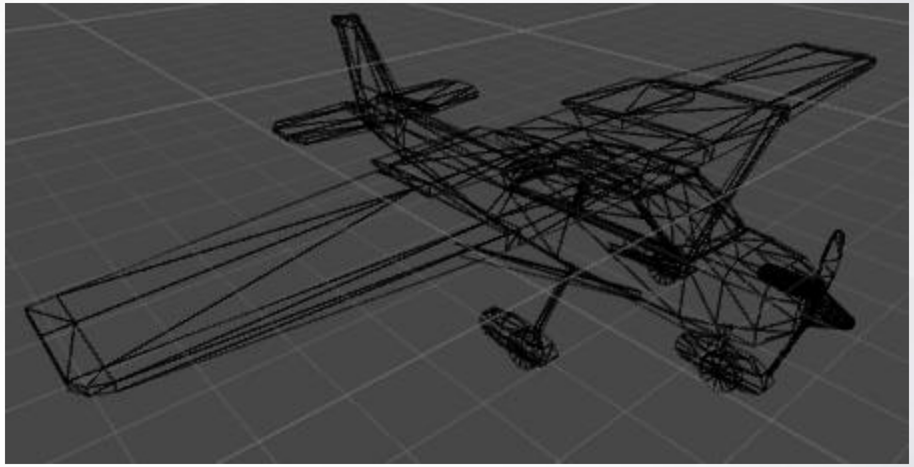


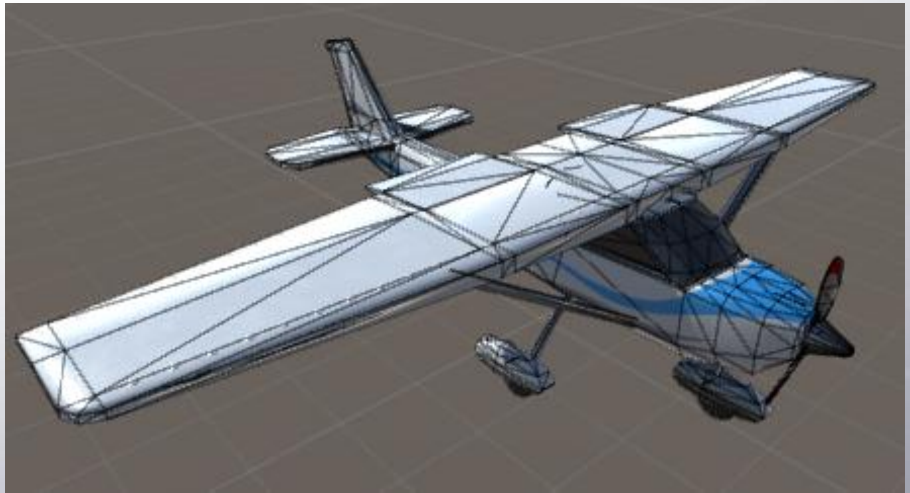
What's Wrong?

1. There's a bad UV
2. There's a bad normal
3. There's a bad vertex (position)
4. There's a bad triangle (vertex index)
5. There's a bad triangle (winding order)

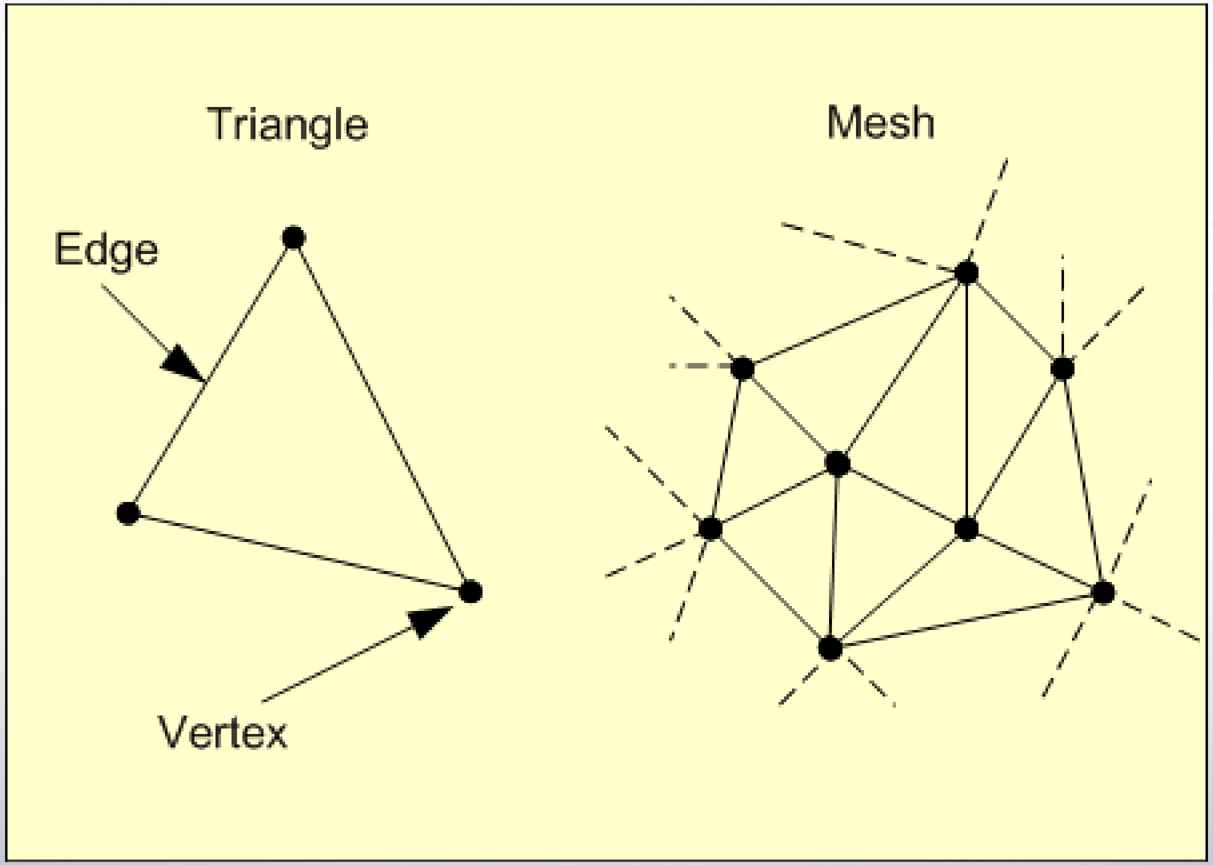
3D Meshes

- A *3D Mesh* is used to store 3D models. It consists of:
  - *Vertices / points (0D):*
    - Represented by *3D vectors*
  - *Line segments* or *edges (1D):*
    - Between two points
  - *Faces* or *polygons / triangles (2D):*
    - Between at least three points that lie in the *same plane* (2-dimensional)





3D Meshes

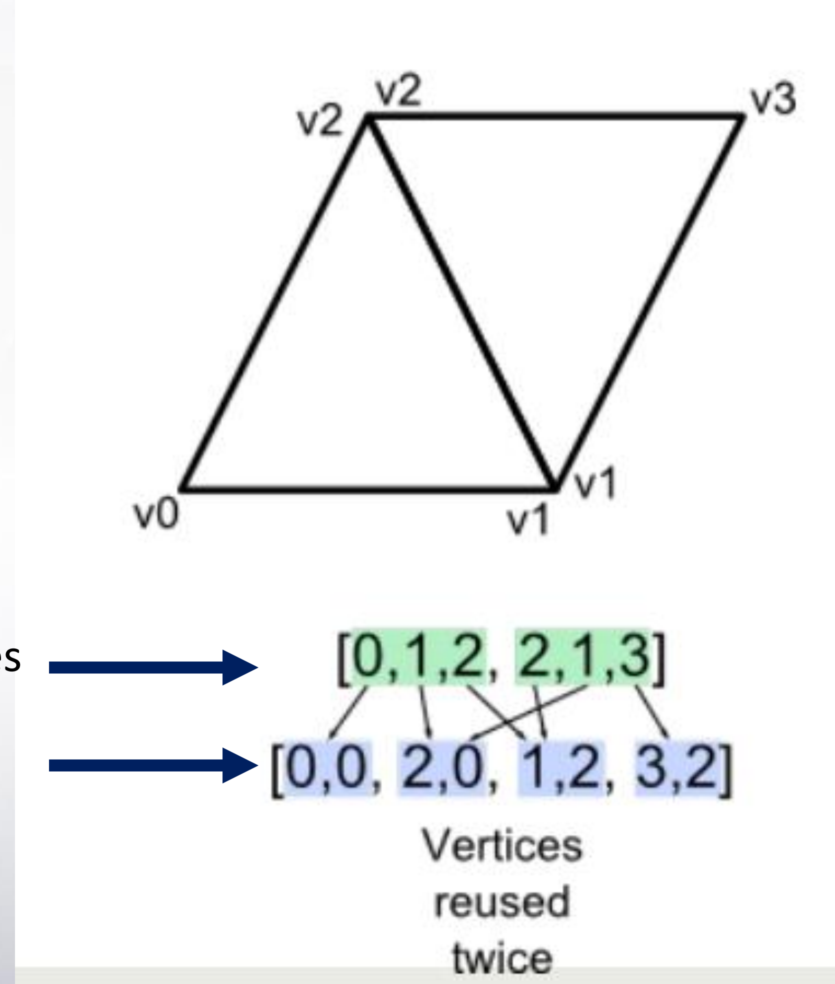


Vertices and triangles

- A Unity (3D) mesh contains an array of *vertices:* each vertex is a Vector3 that contains the local position of a point.
- Unity meshes only contain *triangles* (since you can make any polygon that way)
- Triangles are defined using *vertex indices*.
- The *triangles array* is a list of integers; *triples* correspond to triangles.

Two triples = two triangles

Vertex coordinates



Mesh Creation

- The following code creates a mesh consisting of a single triangle to a game object, and the components to render it:

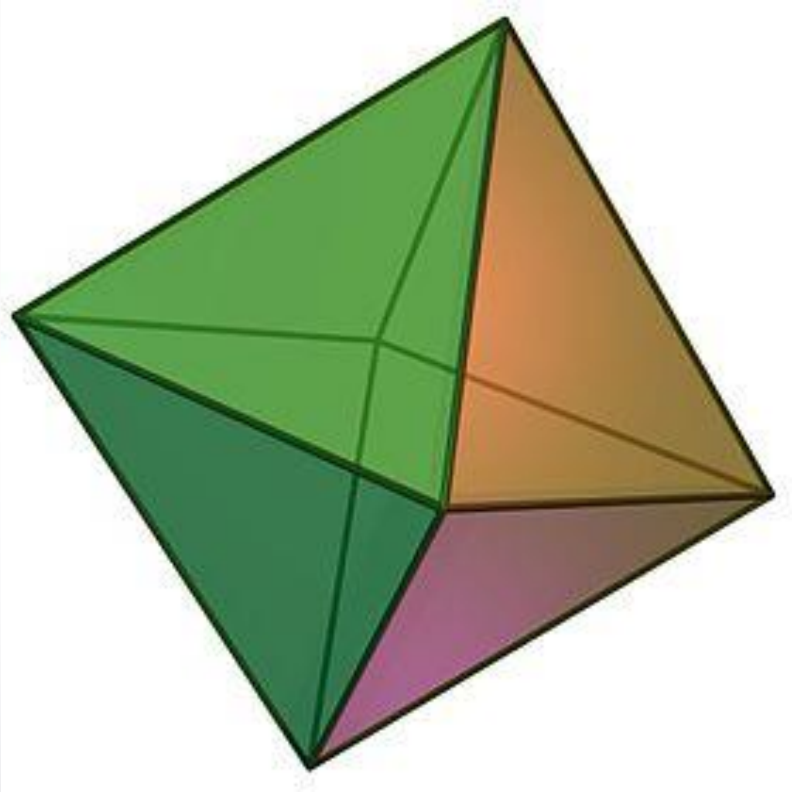
```
using UnityEngine;
using System.Collections;

public class MeshCreateExample : MonoBehaviour {

    void Start() {

        gameObject.AddComponent<MeshFilter>();
        gameObject.AddComponent<MeshRenderer>();

        Mesh mesh = GetComponent<MeshFilter>().mesh;

        mesh.vertices = new Vector3[] {new Vector3(0, 1, 0), new Vector3(1, -1, 0), new Vector3(-1, -1, 0)};
        mesh.triangles = new int[] {0, 1, 2};

    }
}
```

Mesh Builder

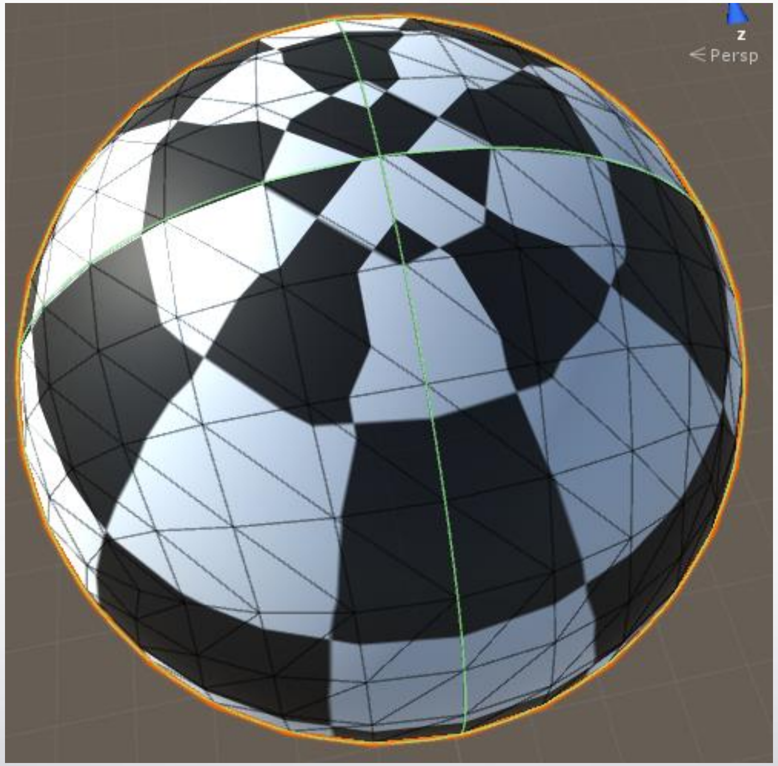
- To make the mesh building process a bit easier, the *MeshBuilder* class is given.
- Main methods:
  - AddVertex
  - AddTriangle
  - CreateMesh

Creating an Octahedron

```
// V2, correct winding:
MeshBuilder builder = new MeshBuilder ();
int v1 = builder.AddVertex (new Vector3 (1, 0, 0));
int v2 = builder.AddVertex (new Vector3 (0, 0, -1));
int v3 = builder.AddVertex (new Vector3 (-1, 0, 0));
int v4 = builder.AddVertex (new Vector3 (0, 0, 1));
int v5 = builder.AddVertex (new Vector3 (0, 1, 0));
int v6 = builder.AddVertex (new Vector3 (0, -1, 0));

// top:
builder.AddTriangle (v1, v2, v5);
builder.AddTriangle (v2, v3, v5);
builder.AddTriangle (v3, v4, v5);
builder.AddTriangle (v4, v1, v5);

// bottom:
builder.AddTriangle (v1, v6, v2);
builder.AddTriangle (v2, v6, v3);
builder.AddTriangle (v3, v6, v4);
builder.AddTriangle (v4, v6, v1);

GetComponent<MeshFilter>().mesh = builder.CreateMesh ();
```

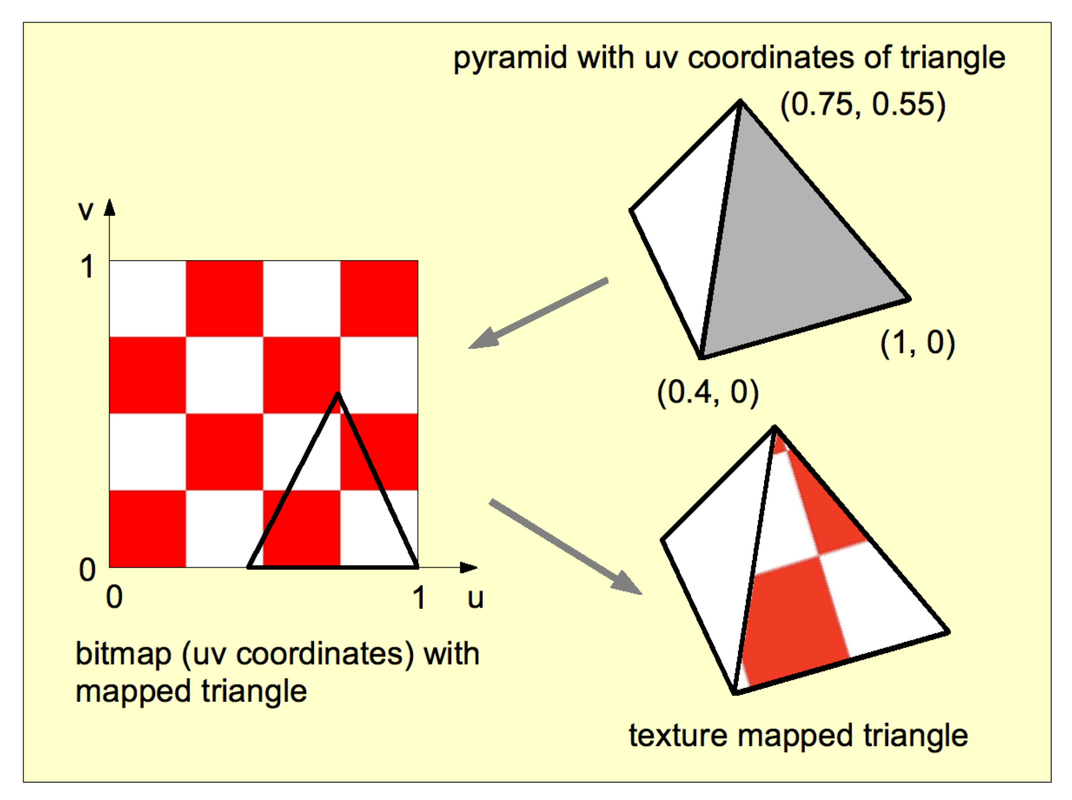


UVs and Normals



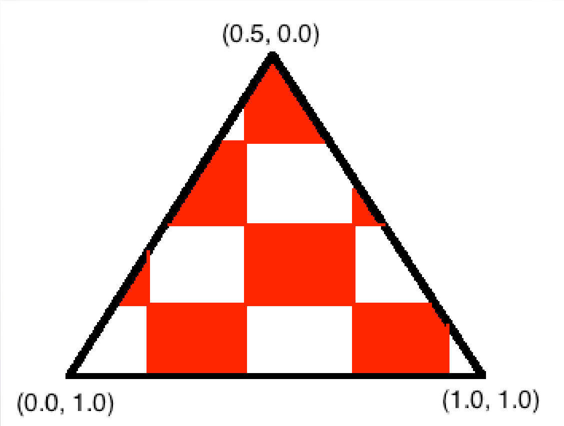
Adding a texture

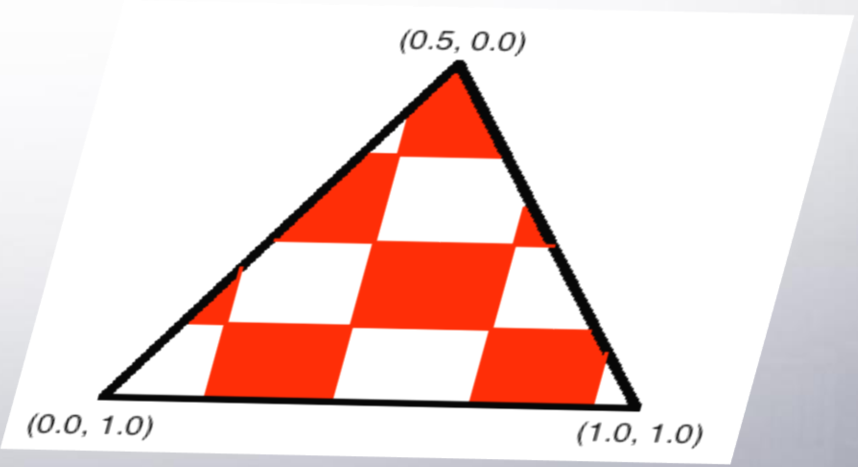
- For every vertex, we need to define what part of the texture corresponds to it. This is not an automated process; we need to define this ourselves.



UV-coordinate

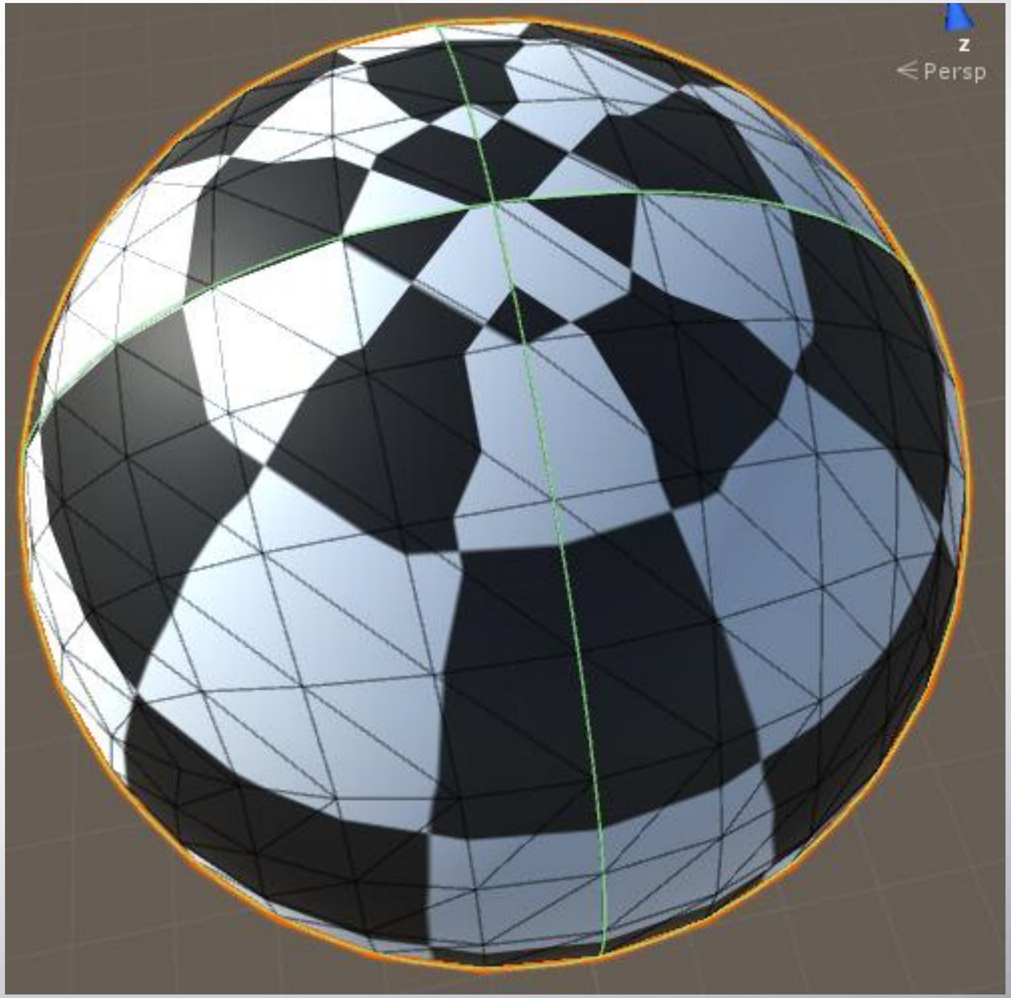
- For every vertex, we must give a *uv-coordinate:* a 2D vector, usually with values between 0 and 1.
- This refers to the texture coordinates (last lecture)
- When rendering the triangle, the uv-coordinates are *interpolated* between the corner values:





- Texturing is hard: here is Unity's sphere with a checkerboard texture:
- Conclusion: in general, mesh, uvs and texture must be designed together!

- Fortunately, for our shape types (buildings are usually not very round), it's less of a problem



UV-coordinates

- UV-coordinates can be outside of the 0-1 range, in that case the texture is *wrapped around* (by default)
- See the *MeshModification scene,* and the *TextureCycle* script

- Let's add uvs to the octahedron
- The MeshBuilder's AddVertex method takes a uv as second parameter

Creating an Octahedron V2

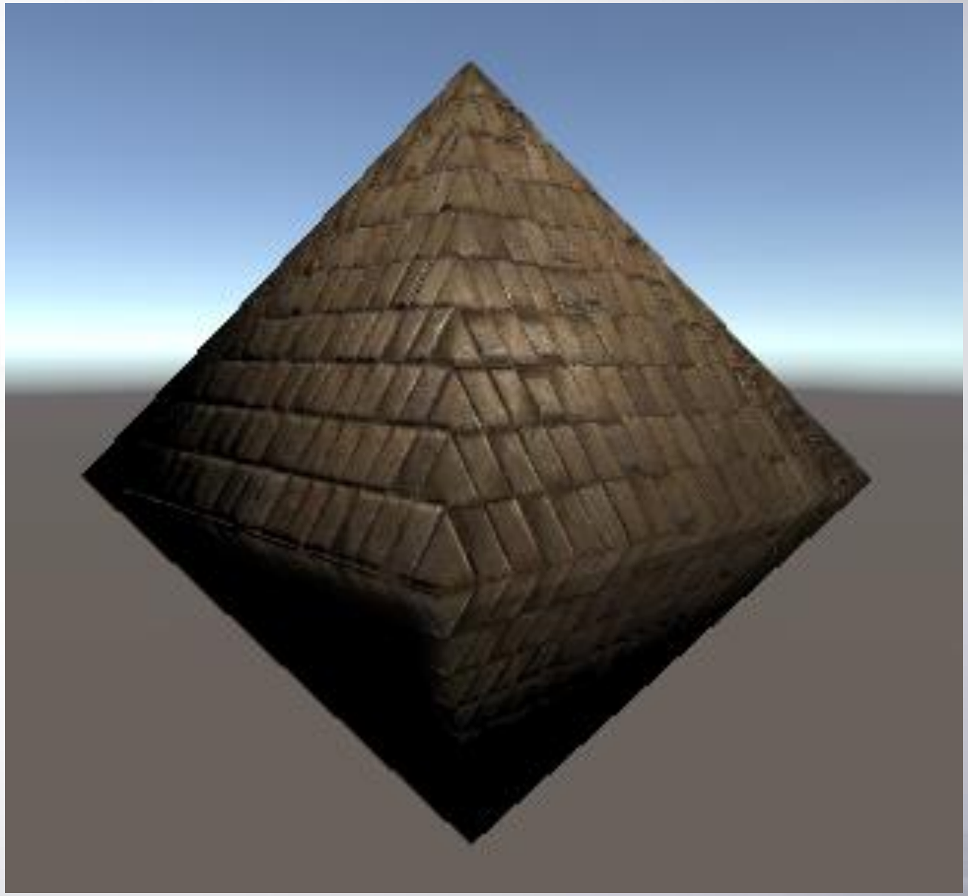
```
// V3, with uvs:
MeshBuilder builder = new MeshBuilder ();
int v1 = builder.AddVertex (new Vector3 (1, 0, 0), new Vector2(0,0));
int v2 = builder.AddVertex (new Vector3 (0, 0, -1), new Vector2(0,1));
int v3 = builder.AddVertex (new Vector3 (-1, 0, 0), new Vector2(1,1));
int v4 = builder.AddVertex (new Vector3 (0, 0, 1), new Vector2(1,0));
int v5 = builder.AddVertex (new Vector3 (0, 1, 0), new Vector2(0.5f,0.5f));
int v6 = builder.AddVertex (new Vector3 (0, -1, 0), new Vector2(0.5f,0.5f));

// top:
builder.AddTriangle (v1, v2, v5);
builder.AddTriangle (v2, v3, v5);
builder.AddTriangle (v3, v4, v5);
builder.AddTriangle (v4, v1, v5);

// bottom:
builder.AddTriangle (v1, v6, v2);
builder.AddTriangle (v2, v6, v3);
builder.AddTriangle (v3, v6, v4);
builder.AddTriangle (v4, v6, v1);

GetComponent<MeshFilter>().mesh = builder.CreateMesh ();
```

Result

- Nice!
- Though on the bottom, the texture is mirrored.
- Preventing that requires more than 6 vertices!

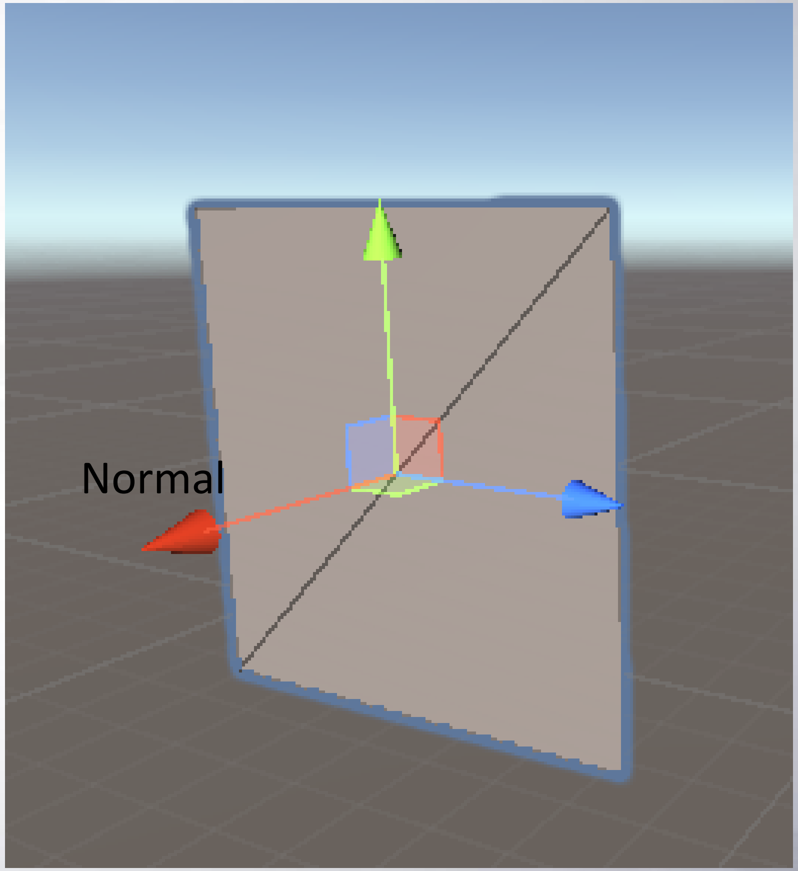
Q: And why is the lighting so weird?

A: That’s related to *normals*



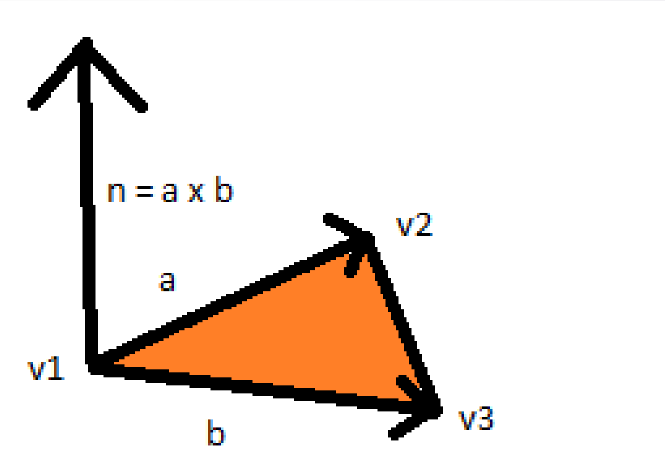
Normal Vectors - Definition

- In 2D:
  - Every line / line segment / vector has a *normal vector*
  - It is perpendicular to the original vector (90 degrees)
  - Computation: (-y,x)
- In 3D:
  - Every triangle / polygon / face has a *normal vector*
  - It is perpendicular to the original face
  - So it is perpendicular to all vectors / line segments that lie in that face, including the edges of the polygon



Calculating Normals in 3D

- For a triangle on vertices v1, v2, v3 (in clockwise order!):
- a = v2-v1 (the vector from v1 to v2)
- b = v3-v1 (the vector from v1 to v3)
- Normal = Vector3.Cross(a,b)
- This is the *cross product* (See 3D Math)



Vertex Normals

- In a mesh, every *vertex* has a normal (?!)
- *Demo: MeshBreathe* script applied to Unity cube / sphere
- *Conclusions:*
  - A Unity cube has 3x8 = 24 vertices after all, each with different normals!
  - However, the Unity sphere does seem to have *shared vertices:* each vertex is part of 4 to 6 different triangles
  - Why? → next slide
- Unity can compute vertex normals for you using *RecalculateNormals* (see *MeshBuilder)*
  - How? → Later

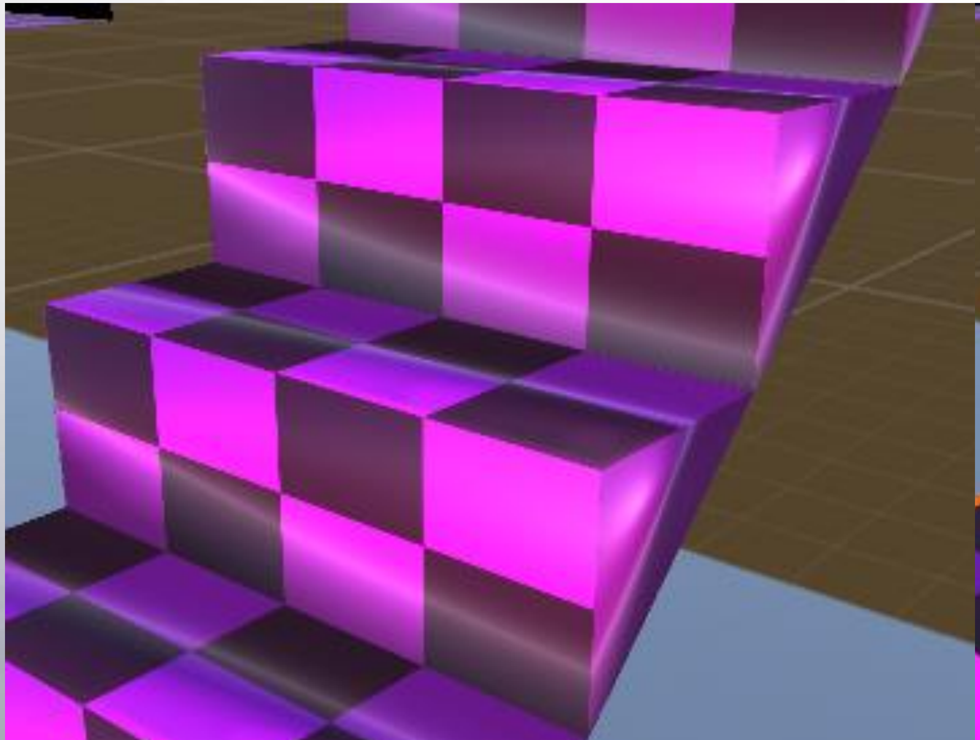
Procedural Staircase

- In the *MeshGeneration* scene, an (incorrect) staircase mesh is generated.
- Next to it, the correct mesh is shown
- Todo:
  - Fix the triangle winding order (one triangle is “facing inside”)
  - Fix the UVs
  - Try to avoid *shared vertices*
  - Try to discover: how does this change the lighting / normals?
  - Try to discover: how does Unity compute vertex normals?
  - Possibly: Add left / right / back side to the steps
- Tip: *First make a drawing using pen and paper!!!*

Shared Vertices, Normals & Lighting

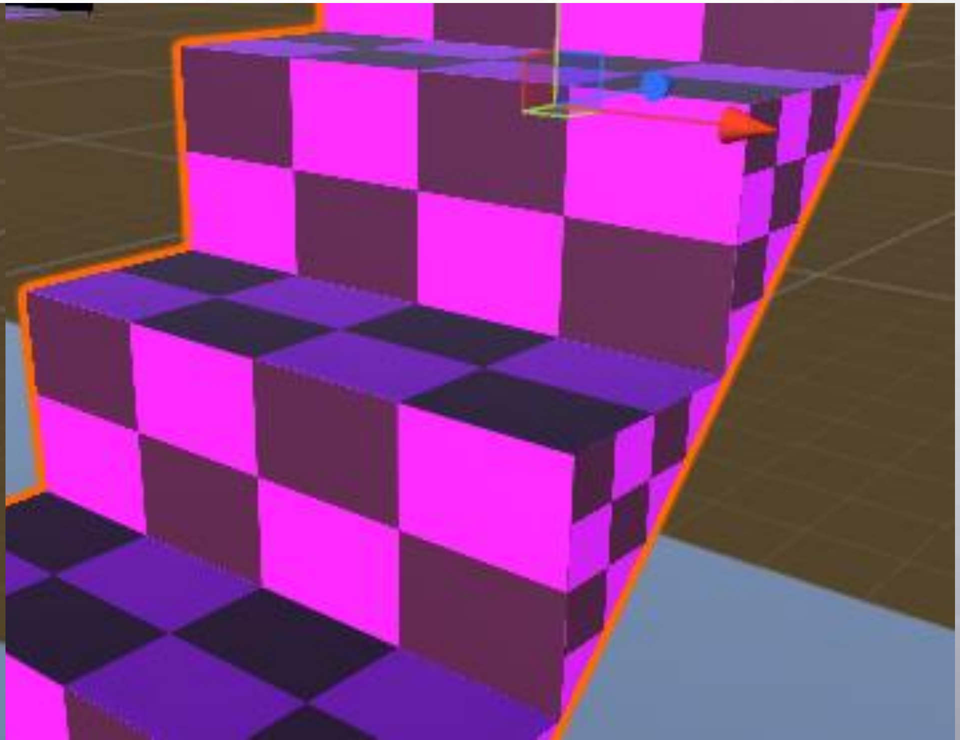
Shared vertices:

(6 vert. per step)



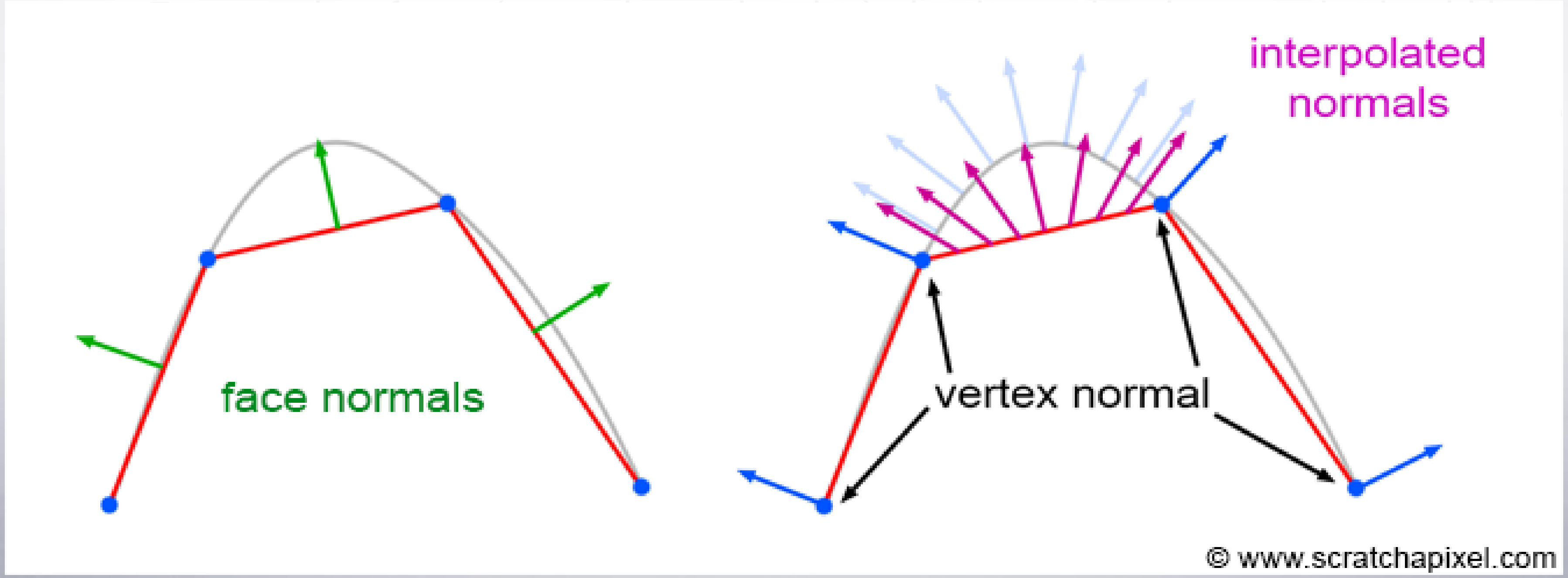
No shared vertices:

(18 vert. per step)



Vertex Normals - Application

- When drawing a triangle, Unity interpolates the vertex normals too!
- This is used for lighting computation (demo)
- *Conclusion:*



Vertex Normals - Application

- *Conclusion:* to get correct lighting, …:
  - For *round surfaces* (e.g. sphere triangles), the three vertex normals of the triangle should be different
  - For *flat surfaces* (e.g. cube sides), the three vertex normals of the triangle should be the same
  - The Unity primitives have correct normals

Vertex Normals - Calculation

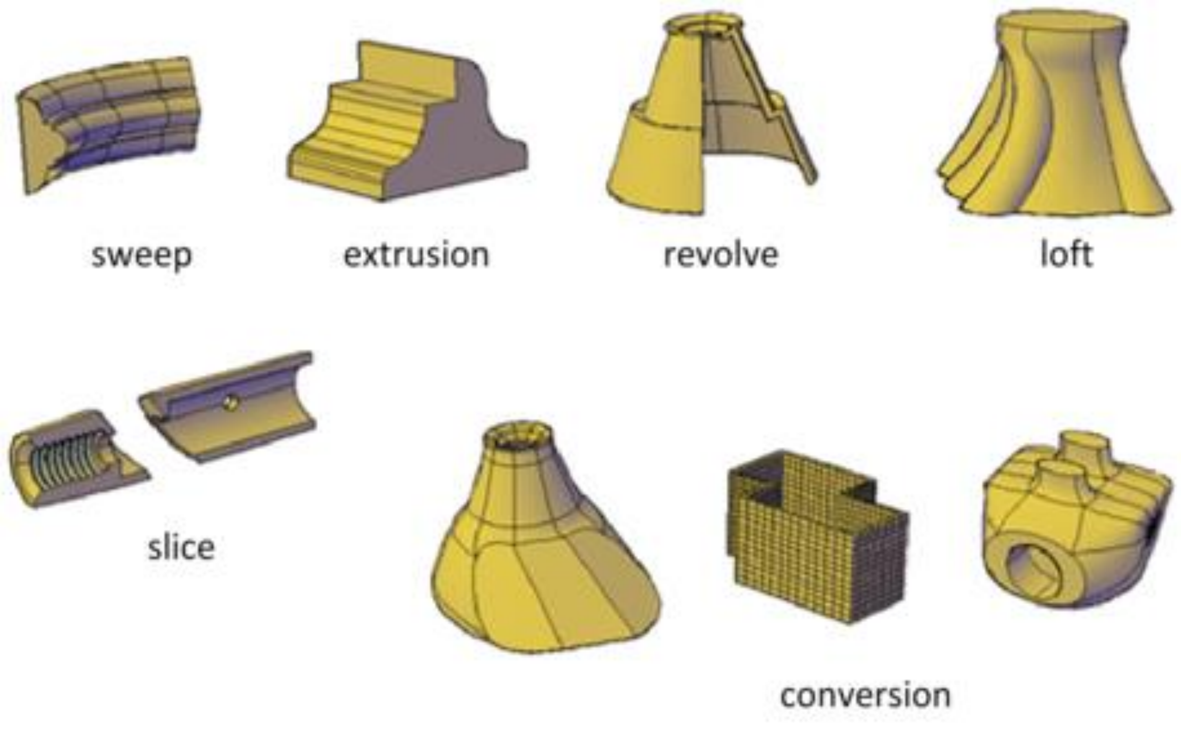
- Apparently, the *RecalculateNormals* method does the following:
  - Compute the *face normal* for every triangle (using cross product)
  - The *vertex normal* is the *average* of all face normals of incident faces

- (“Proof by demo” – using *MeshBreathe* and *Octahedron*)
- Conclusion:
  - For flat surfaces, avoid shared vertices
  - For rounded surfaces, use shared normals
  - For full lighting customization: *use your own method to define / compute vertex normals*

Lathe

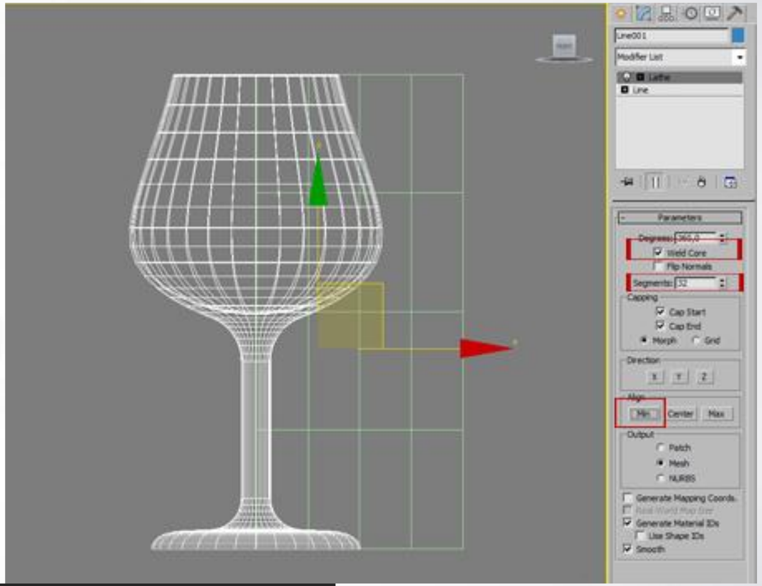
Modeling Tools

- Modeling tools use various algorithms to *modify* meshes, or generate complex meshes from simple meshes (or point sets)



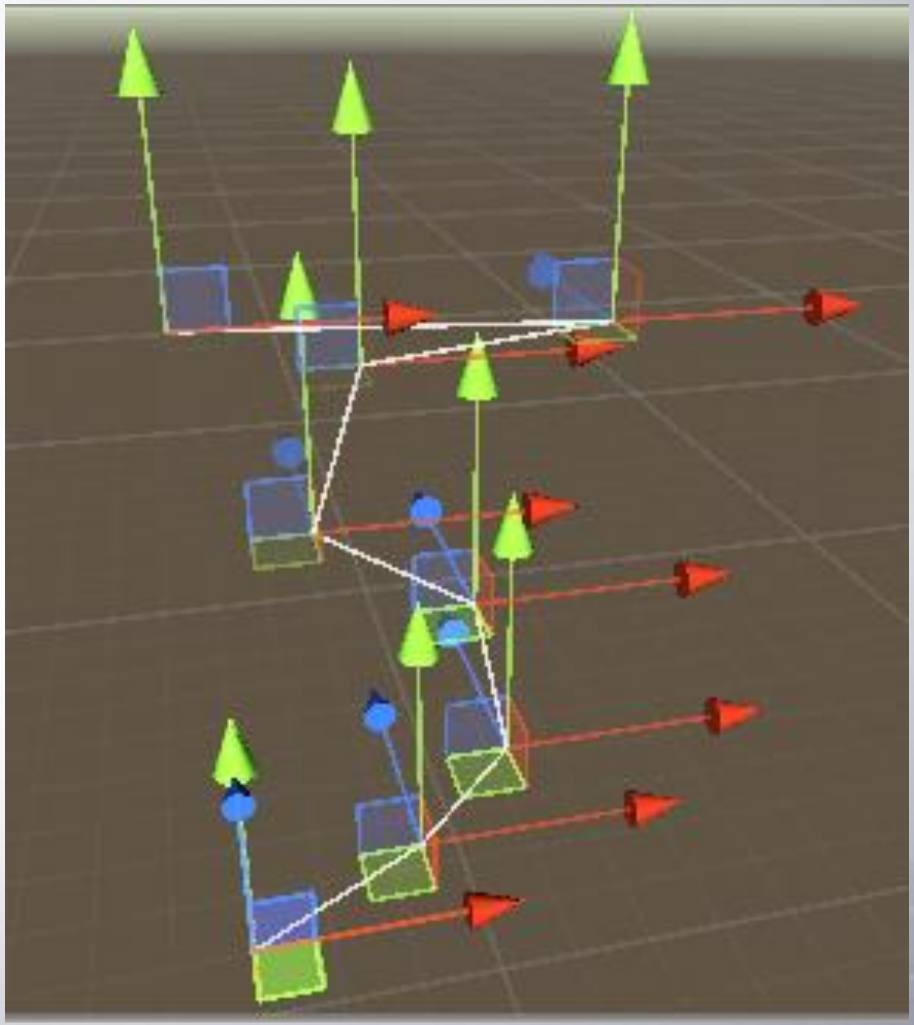
Lathe

- The *lathe* tool takes a *spline* or *curve* and rotates it around an axis.
- *Spline:* sequence of 2D points
- You can use it for example to create: pawns, glasses, rockets, vases and round buildings.



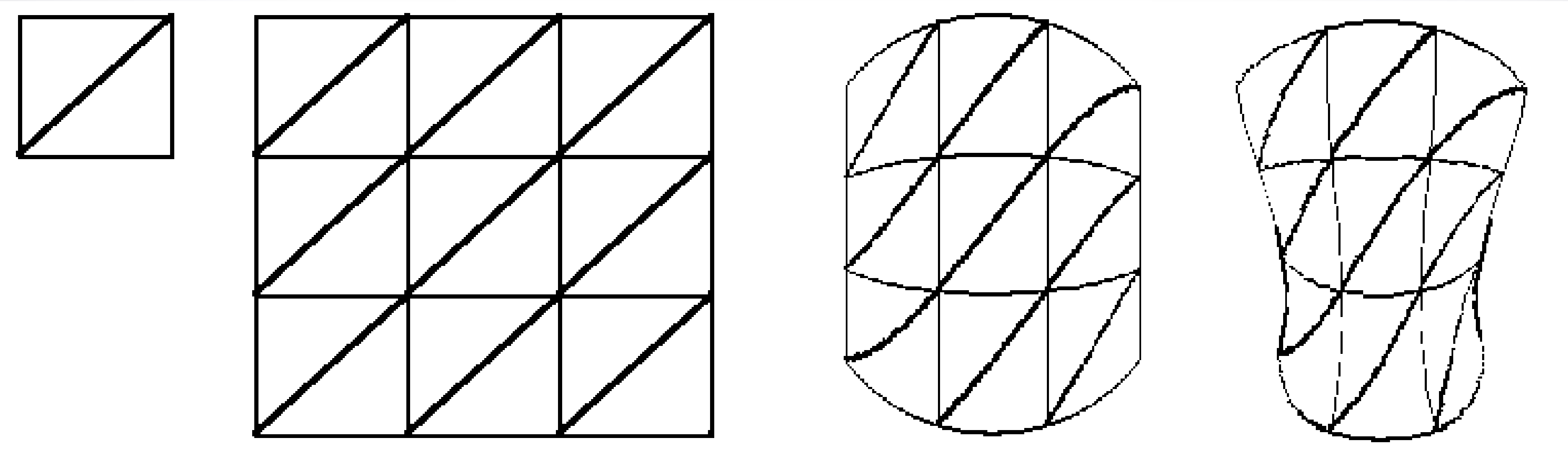
Curve Component + Editor Tooling

- Scripts in handout:
  - Curve.cs → Contains a number of 3D points
  - CurveEditor.cs → Creates scene editor tooling / gizmo's
  - MeshCreator.cs → An abstract superclass: mesh creators take a curve as input, and create a mesh from it
  - Specific MeshCreators:
    - LatheSpline.cs
    - Extrude.cs
    - WarpMeshAlongSpline.cs
- Usage:
  - Add a *Curve* component and one of the *MeshCreator* components to a game object, with a *MeshFilter* and *MeshRenderer* component.
  - In the scene editor, change the curve using the gizmo's
- For an explanation of how Editor tooling works (CurveEditor): see the week 3 lecture, or see the Unity documentation, e.g. https://docs.unity3d.com/ScriptReference/Handles.PositionHandle.html



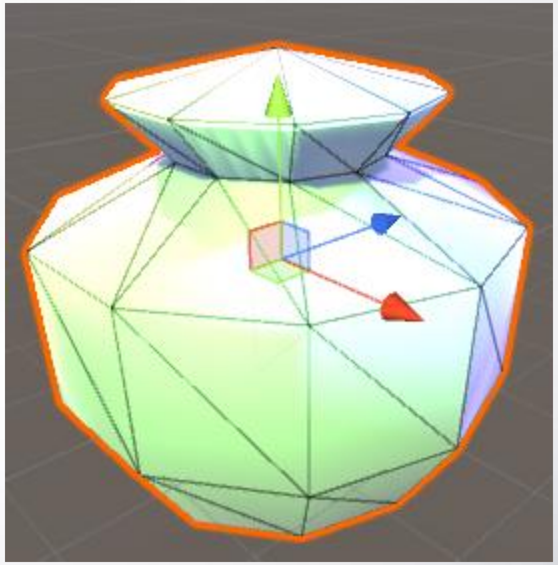
Building a Lathe Algorithm

- A lathe operation can be seen as a warped cylinder.
- A cylinder can be seen as a warped plane.
- A plane can be seen as a collection of quads.

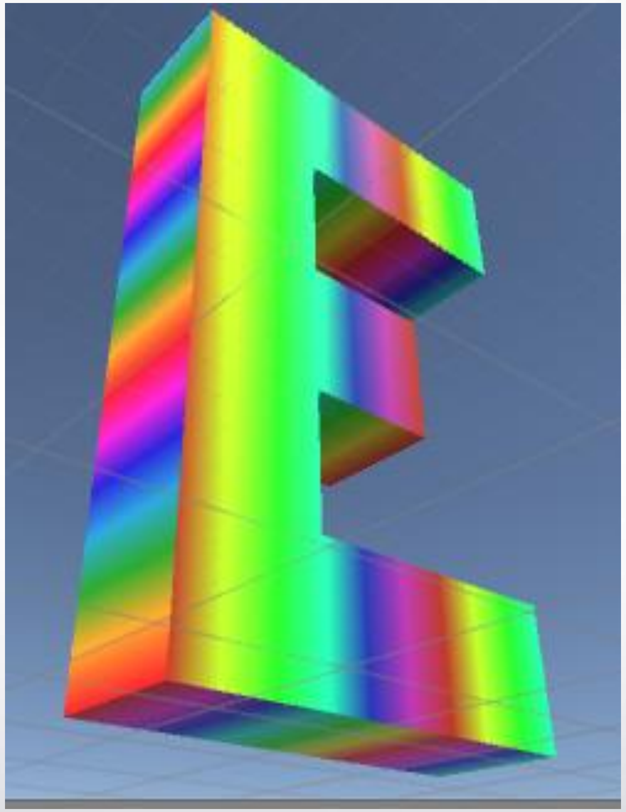


LatheSpline.cs

- Let's try out + study the script *LatheSpline* from the given Unity project
- Two nested for loops to generate *vertices:* creating one *circle* for every spline vertex
- Spline vertex x-coordinate gives circle radius, y-coordinate gives circle height
- Two nested for loops to generate quads
- Use Quaternion.Euler to rotate points around the y-axis

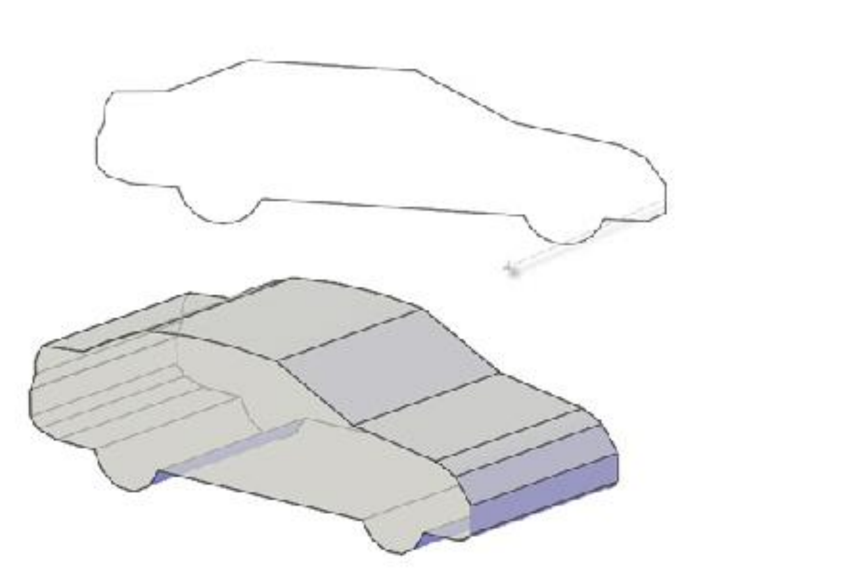


Extrude



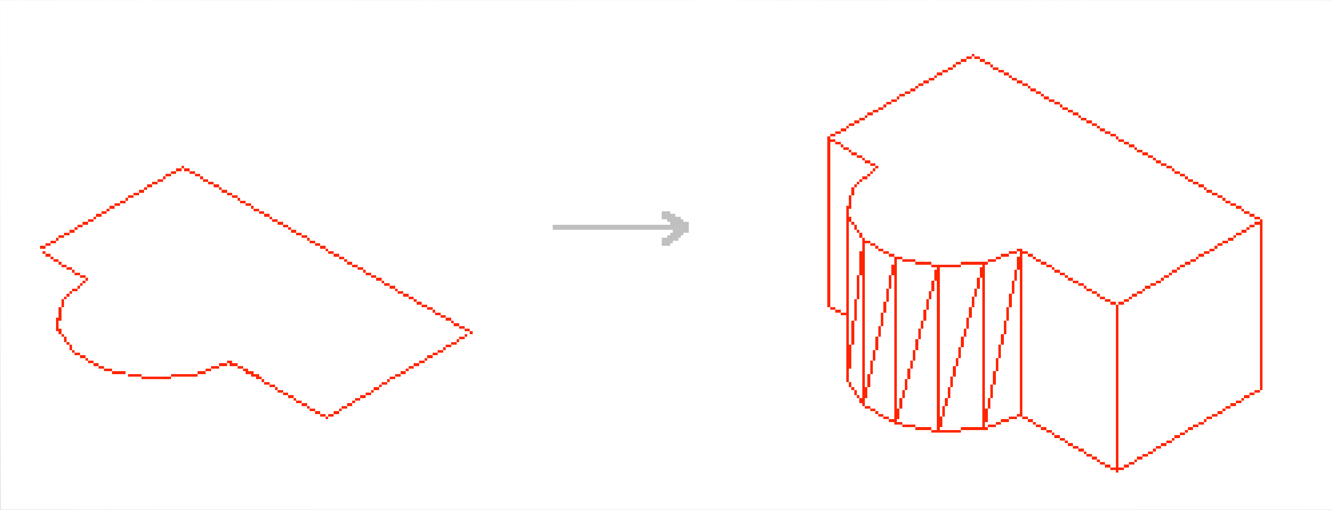
Extrude Operation

- The *extrude* operation extends a face of a mesh in its normal direction.
- Or: starting with a 2D *polygon,* create a 3D mesh from it, with given width



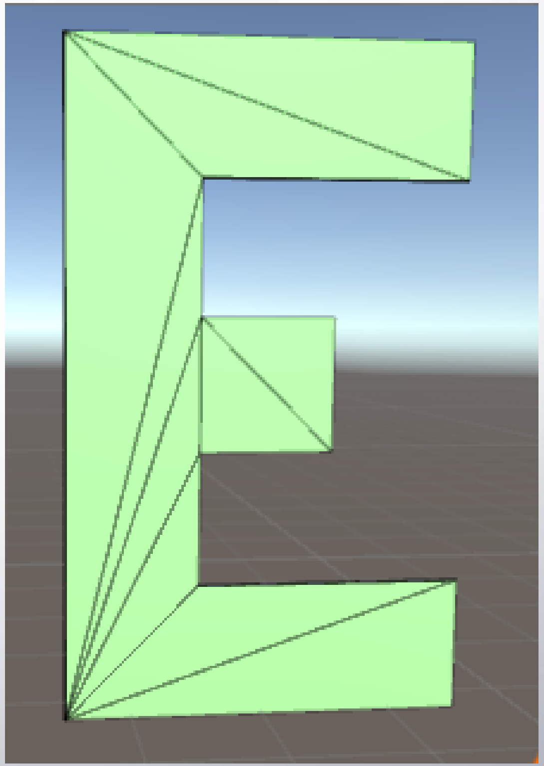
Application

- The extrude operation is particularly useful to create architectural items / procedural cities!
- Input: top down map → Output: building
- See the *HorizontalCurves* scene for some examples



Triangulation

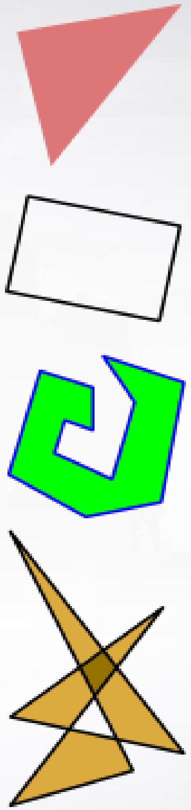
- The most challenging part of this operation is filling the polygon. We need to split the polygon into triangles.
- This process is called *triangulation.*
- *(Demo in Unity)*



Polygons

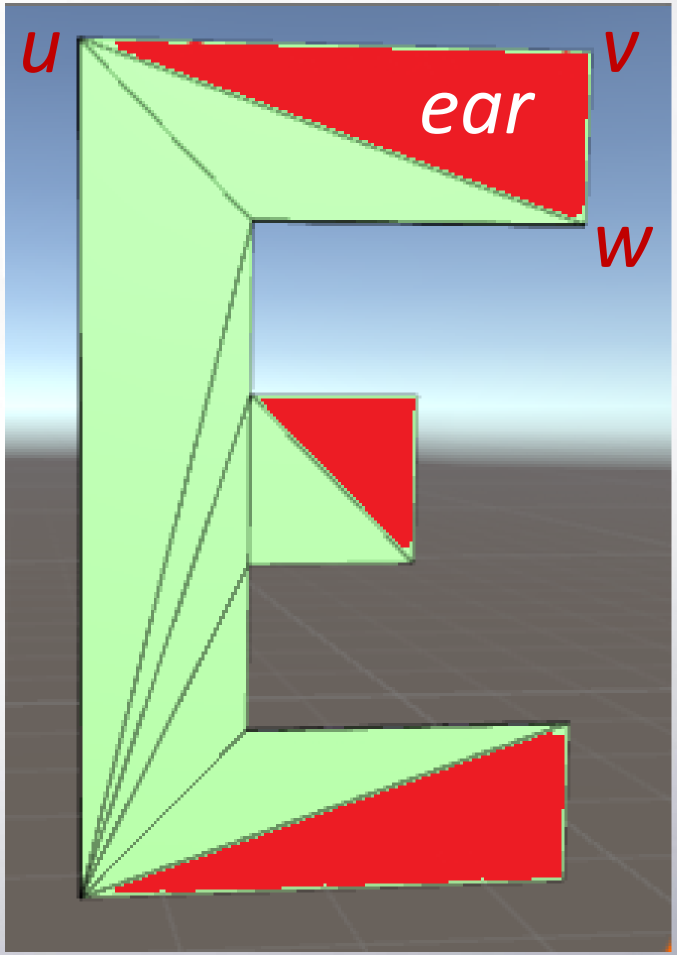
Recall:

- A *polygon* is a 2-dimensional shape consisting of a closed chain of line segments
- In code, we represent it using a sequence of 2D points (vectors)
  - *Assumption* for our algorithm*:* these points are given in *clockwise order*
- A polygon is *simple* if no line segments overlap

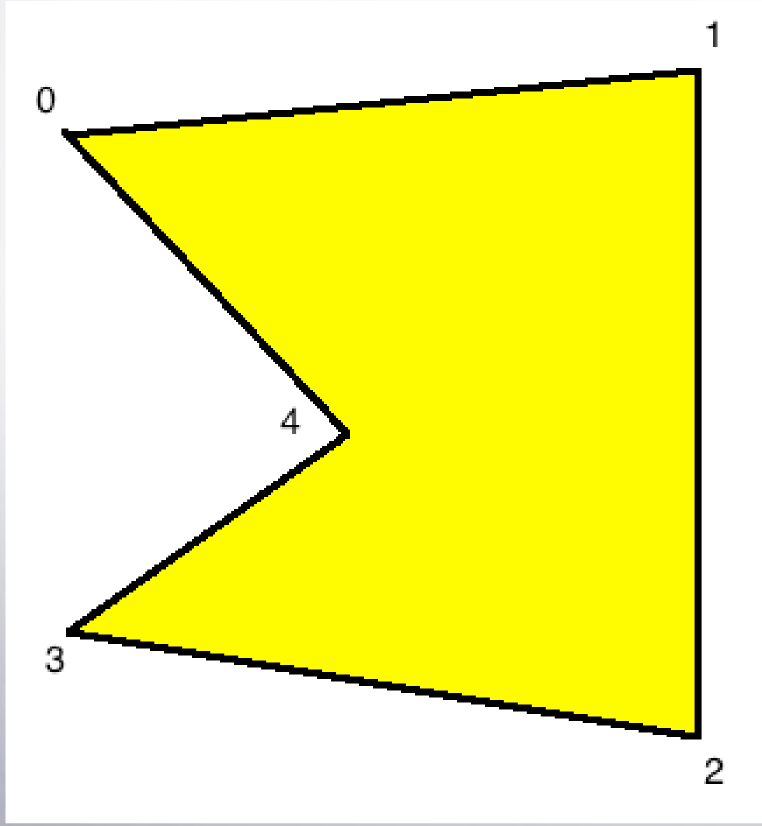


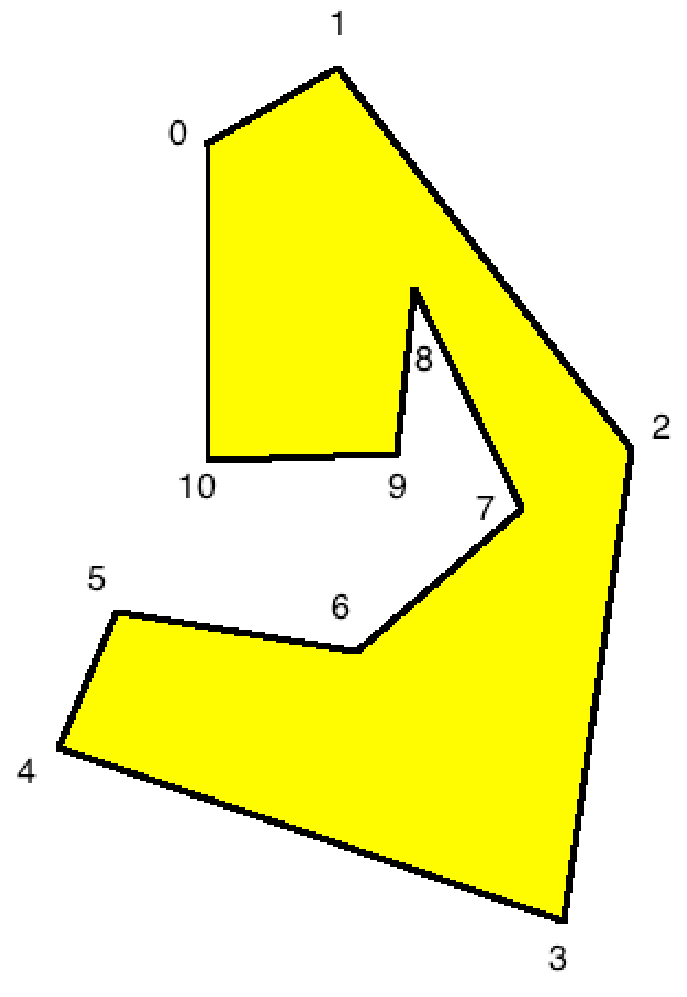
Ears

- Polygon points are called *neighbors* if they are consecutive / joined by a polygon edge
- *Observation:* Every point *v* of the polygon forms a *triangle* together with its two *neighbors u* and *w.*
- This triangle is called an *ear* if it is entirely inside the polygon
- So in particular: *u,v,w* are in *clockwise order*

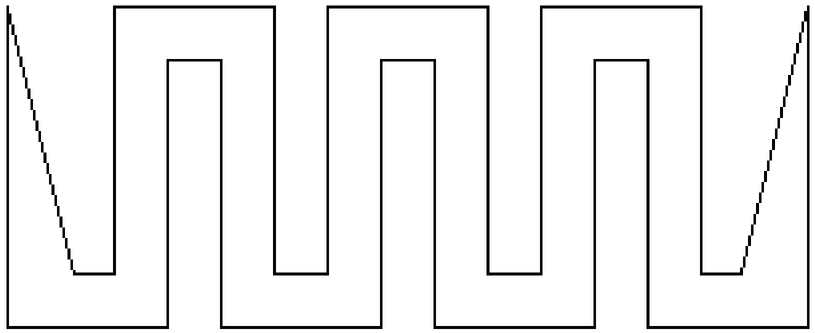


Ears - example

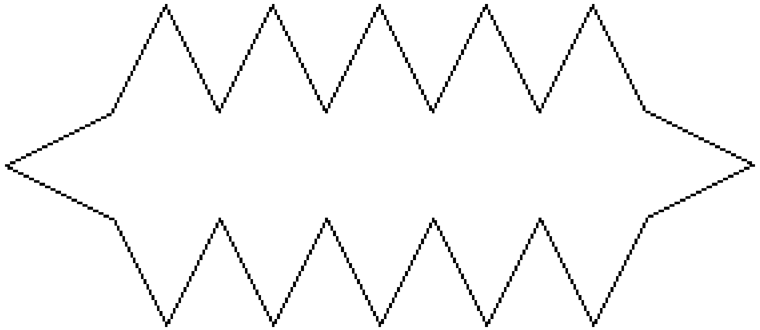




Ears - example



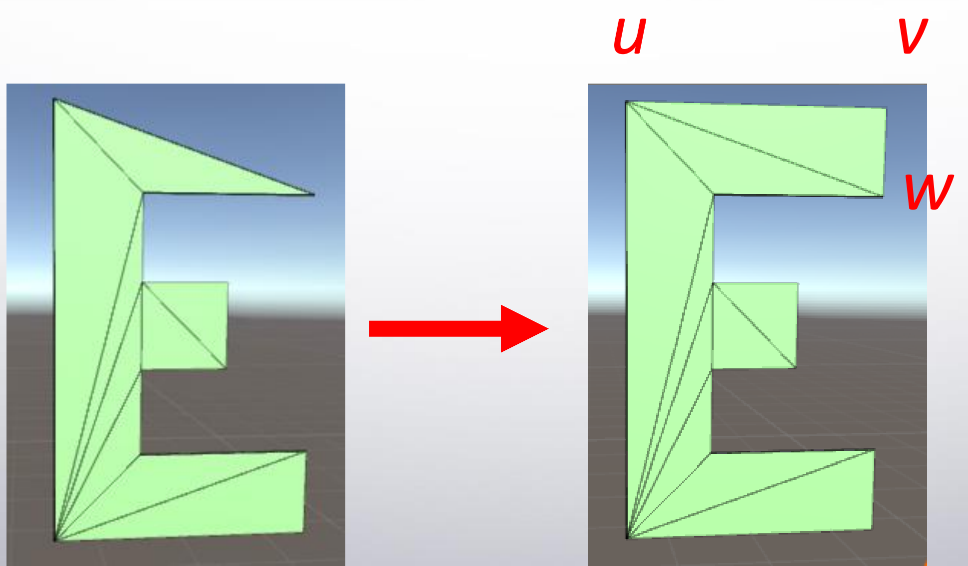
A simple polygon with only 2 ears.



A simple polygon with many ears (12 ears).

Triangulation Method

- *Key observation:* if we can find an *ear u,v,w* of polygon *P,* then we can:
  - Create a smaller polygon *P’* by removing *v* from the point sequence, such that *P’* is *still simple and labeled in clockwise order*
  - Combine a triangulation of *P’* with triangle *u,v,w* (the ear) to obtain a triangulation of *P*



Good News

Two Ears Theorem:

Every simple polygon with more than three vertices has at least two ears.

History and proof [edit]

The two ears theorem is often attributed to a 1975 paper by Gary H. Meisters, from which the "ear" terminology originated.[4] However, the theorem was proved earlier by Max Dehn (circa 1899) as part of a proof of the Jordan curve theorem. To prove the theorem, Dehn observes that every polygon has at least three convex vertices. If one of these vertices, v, is not an ear, then it can be connected by a diagonal to another vertex x inside the triangle uvw formed by v and its two neighbors; x can be chosen to be the vertex within this triangle that is farthest from line uw. This diagonal decomposes the polygon into two smaller polygons, and repeated decomposition by ears and diagonals eventually produces a triangulation of the whole polygon, from which an ear can be found as a leaf of the dual tree.[5]

Algorithm

- While there are at least three points in the list:
  - Find an ear *uvw.*
  - Remove *v* from the point list
  - Add *uvw* to the list of triangles
- The *Extrude* script implements this
- You need to add the *ear checking part* yourself!
- *Hint:* two very useful methods are given on the bottom.

Summary

TODAY:

- Mesh basics: vertices, triangles
- uvs and normals
- Lathe & curves
- Extrude & triangulation

NEXT WEEK:

- Warping meshes
- Texturing procedural meshes
- Assets and scenes