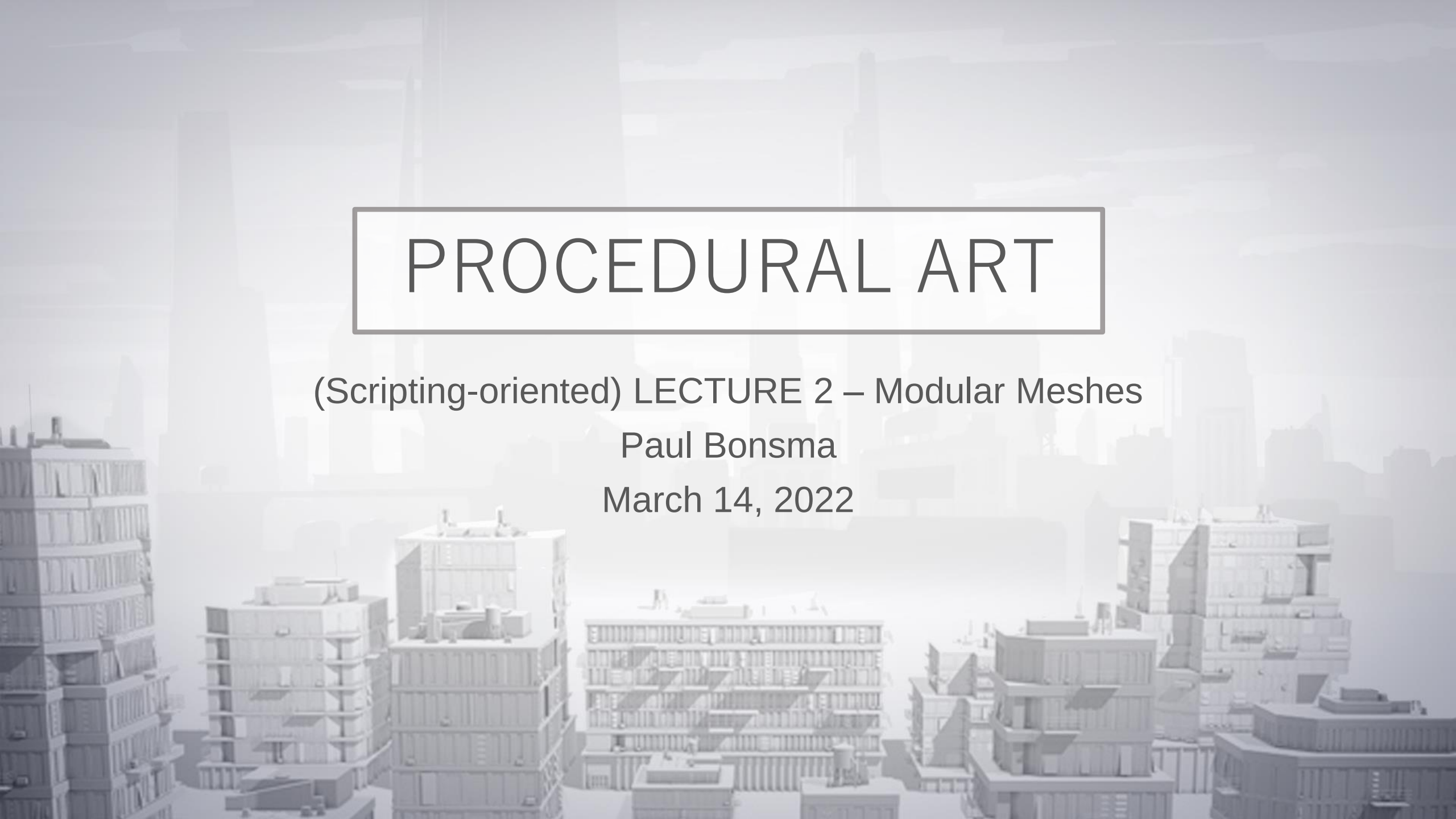


PROCEDURAL ART

(Scripting-oriented) LECTURE 2 – Modular Meshes

Paul Bonsma

March 14, 2022

Why Procedural (City) Generation?

- Infinite variety / infinite replay value
  - →*Run time generation*
  - →Used by infinite runners, roguelike games, etc.

- Tooling / fast creation of large scenes
  - →*Editor mode generation*
  - →Used by any (open world) game / movie that needs a lot of content

Why Procedural (City) Generation?

A more subtle reason:

- Compression / storage space
  - → *It's easier to store a few "random seeds" + some general city outline data than a full scene with millions of polygons!*
  - → Runtime generation, editor time construction
  - → How much *input* does your generation algorithm need?

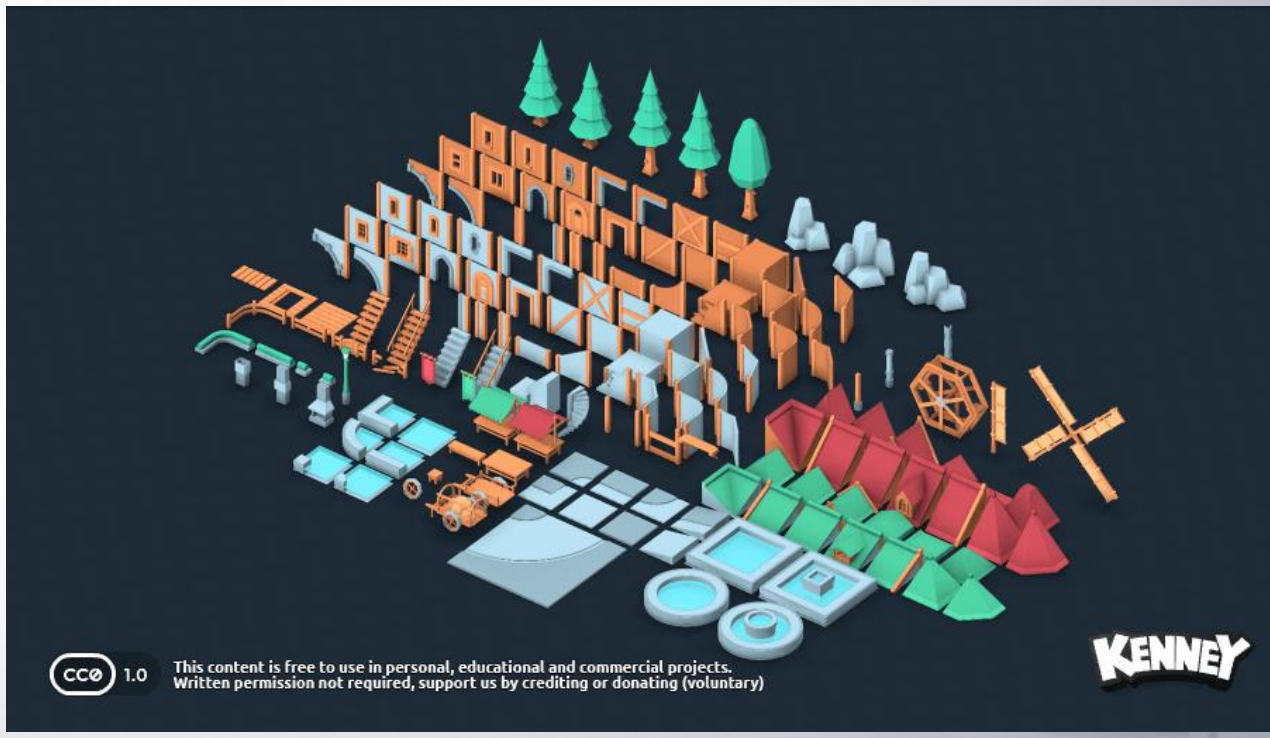
Run Time vs Editor Time Generation

- This lecture:
  - (Almost) everything is generated at runtime, in play mode
  - Generating a full and interesting city completely procedurally this way is hard!
  - For the assignment, you can also have manual parts (such as placing procedural building roots manually in the scene, on a terrain)
  - You can also customize the generated buildings afterwards (hopefully using good editor tools, not manually!)
  - Next week: Unity editor tooling

Today: Modular Meshes

Creating scenes (meshes)…

…from building blocks (modules):



“How does it help me pass the course?”

This lecture + handout will help you directly towards these grading criteria:

- *A wide range of shapes is created procedurally from a smaller range of building blocks.*
- *There is a lot of recognizable variety in the buildings.*
- *Buildings can be customized after generation.*
- *Building parameters can be controlled before generation (e.g. different neighborhoods have different building styles).*

It might help you to achieve these criteria:

- *The resulting structures match the visual research.*
- *Optimizations have been done for real-time efficiency (LOD groups).*
- *Consistent scale is used for size and placement of buildings and other structures.*
- *The overall look matches the chosen theme.*

Outline

- Creating Building Blocks
- Shape Grammars
- Grid Based Methods
- Conclusion

Creating Building Blocks

Step 1: Identify your Building Blocks

- Look at your *visual references*.
  - What kind of *building blocks* do you see?
  - How would you *describe the structure* of this building?

- You would describe this building using words like *stock, roof, wall, window, door, corner*, and describe the relation between them (“a wall contains a sequence of windows”)
- Even though the Kenney pieces are made for a “fantasy town”, they can actually be used very well to create buildings like this!

Creating Building Blocks: General Tips

- Use consistent block *scales*
  - → Kenney's fantasy town: nearly all blocks are 1x1x1, created with tiling in mind
  - → Tip: if you want to have consistent and realistic building scales where 1 unit = 1 meter, and one block is one stock high, 3x3x3 blocks may be better
- Use consistent *rotation* for pieces
  - → Kenney's fantasy town: all wall pieces and corner pieces face the same direction
- Use consistent *pivot points* for pieces
  - → Kenney's fantasy town: nearly all blocks have their pivot point at the bottom center (excellent for rotation around y-axis, the most common operation for such pieces)

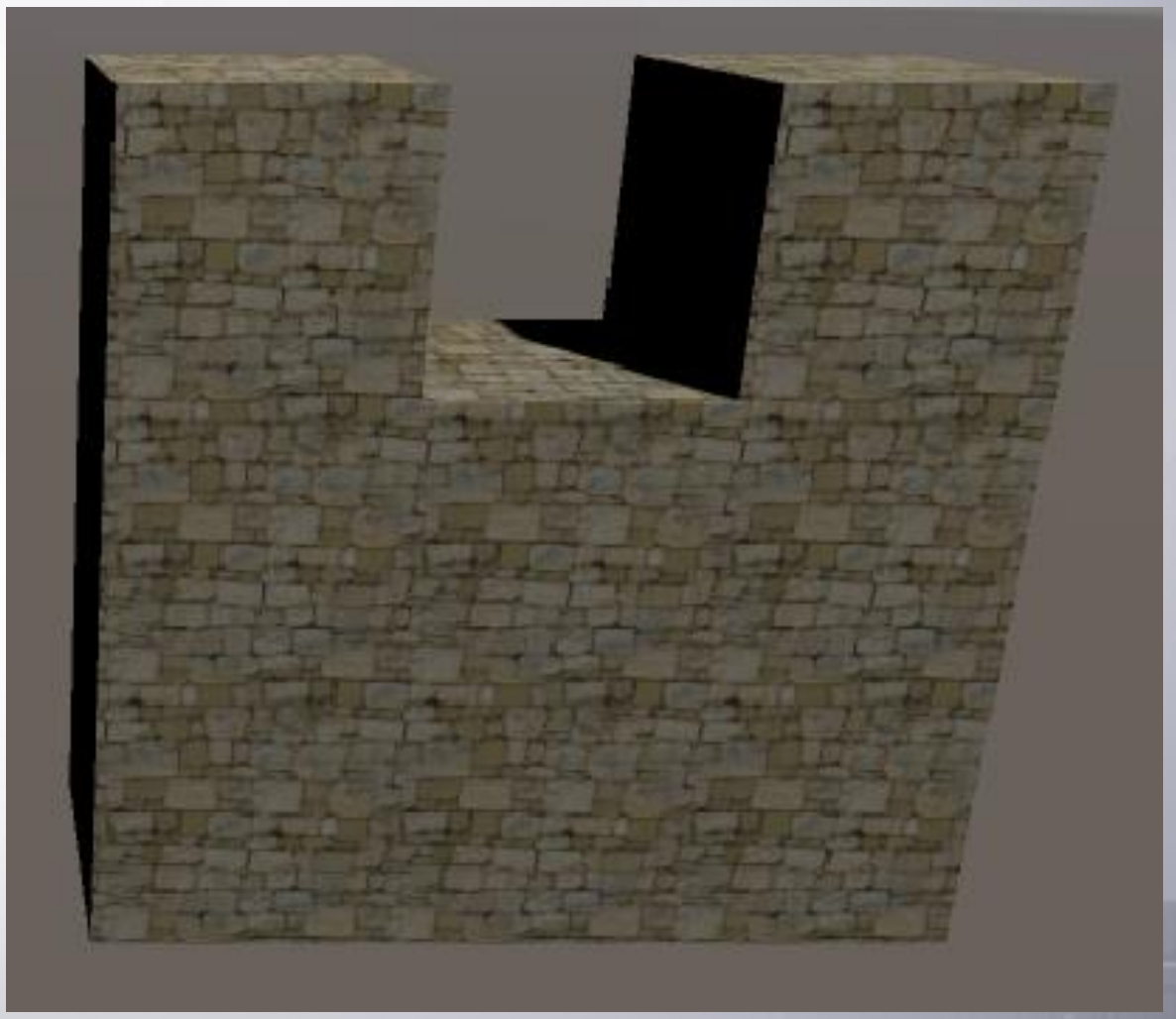
…let's have a look in Unity at the Kenney pieces…

Creating Building Blocks: Textures

- Make sure you use *tileable* textures!
  - → Substance Designer helps! (Art-oriented lecture)

- Set up your uvs with tiling in mind!
  - → This is a tricky problem
  - → Not even the standard Unity cube is perfect!

(Example texture used: by BlueRatDesigns, from opengameart.org)



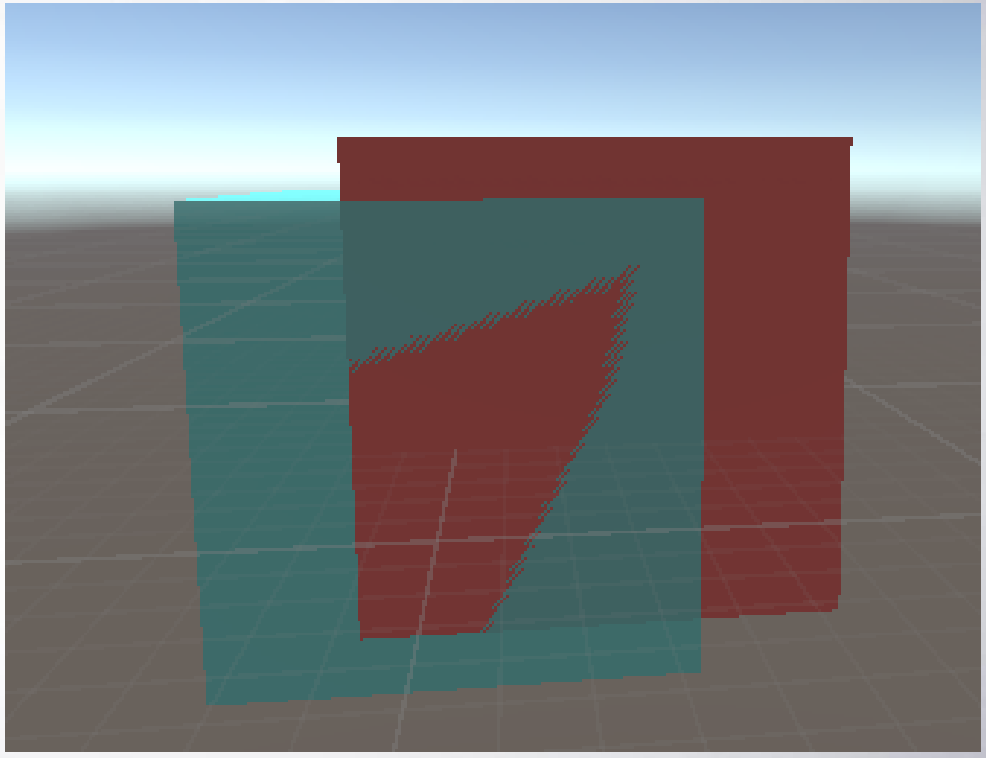
Creating Building Blocks: General Tips 2

- Sometimes it's good to have meshes that are slightly larger than their "box" (see Kenney's roofs), but make sure you *don't have overlapping faces with different materials*

  → This causes *Z-fighting* ➔

- *Optimization:* think about occluded (invisible) mesh parts, when the blocks are tiled!

  → If no one will ever see the inside of a wall, don't model it!



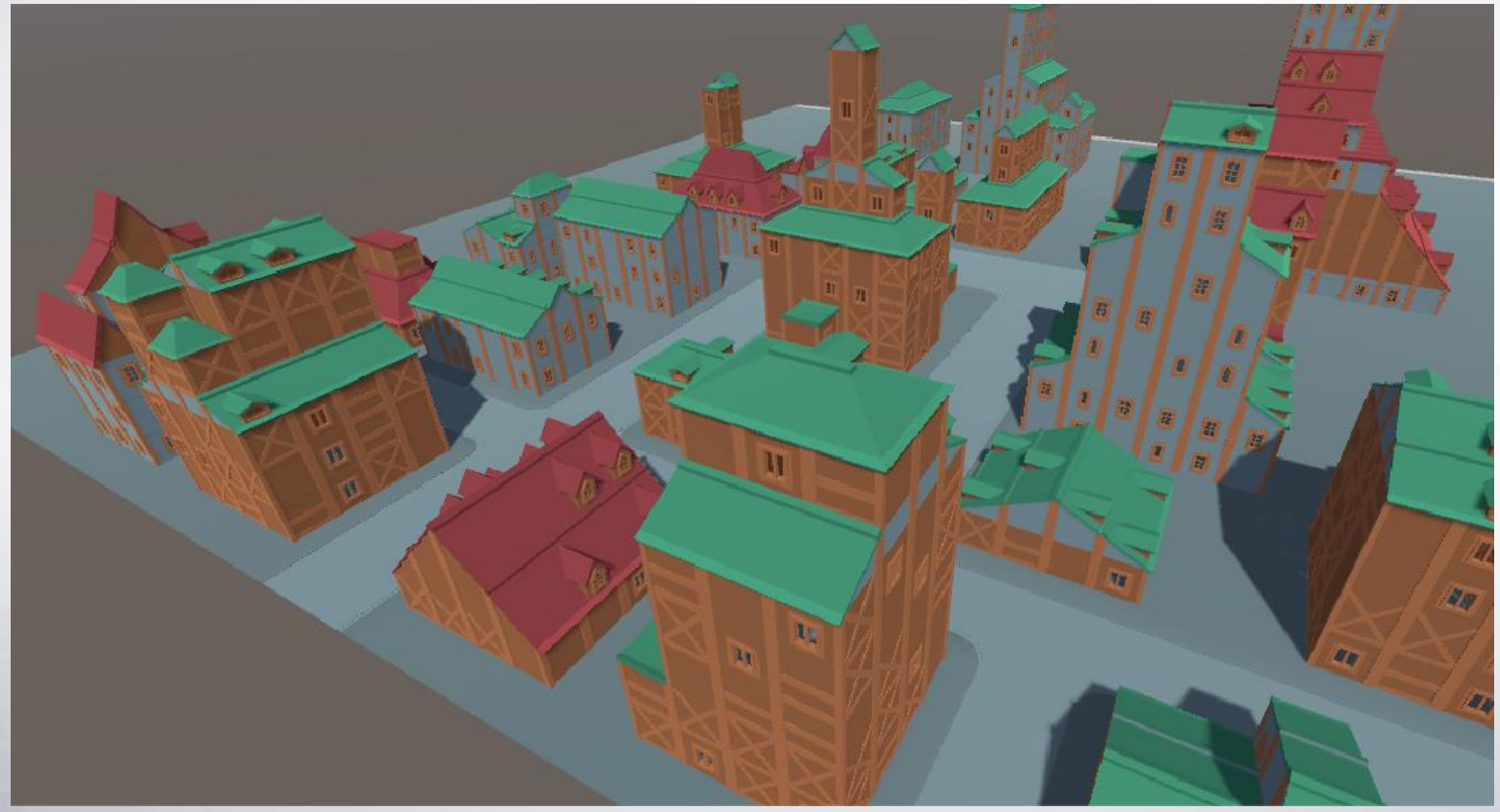
Creating Building Blocks: Most Important Tips!

- Whatever you do, *be consistent!*
- *Fix your conventions* before you start

Example: Kenney Town

- This scene was created in <1 minute:

(After spending many hours on creating tools…)



Technique: *Shape grammars*

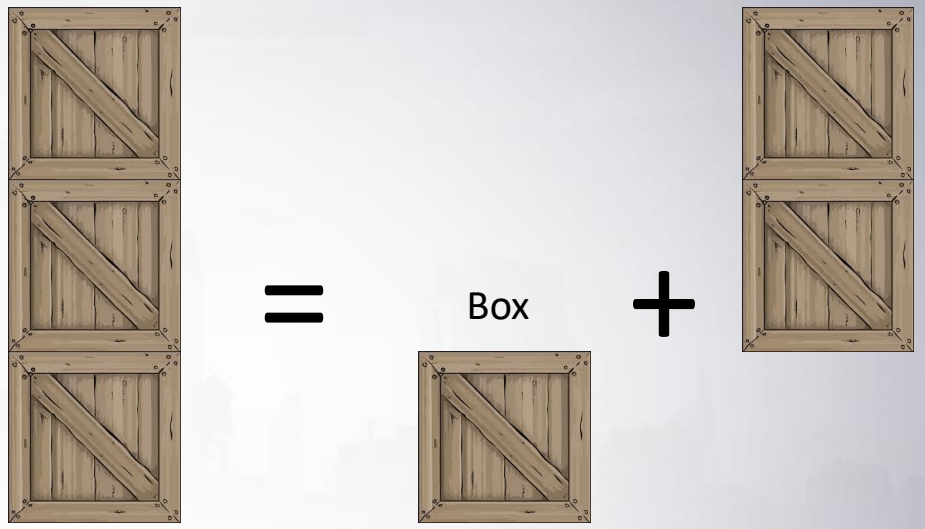
Shape Grammars

Shape Grammars

- *Shape grammars express the hierarchical structure of shapes*

Examples:

- A *stack of boxes* consists of a *box* with a (smaller) *stack of boxes* on top.



- A *tree branch* consists of a *branch piece* with two *smaller branches* at the end, or with a *leaf* at the end.

- A *city block* consists of *buildings* which consist of a *roof* and a number of *stocks,* which consist of *walls,* which consist of *wall pieces* and *windows…*

Symbols

- A shape grammar consists of *symbols*:
  - *Terminal symbols*: concrete objects (e.g. *box*)
  - *Non-terminal symbols*: abstract concepts that consist of multiple objects (e.g. *stack*)

Examples:

- A *stack of boxes* consists of a *box* with a (smaller) *stack of boxes* on top.

- A *tree branch* consists of a *branch piece* with two *smaller branches* at the end, or with a *leaf* at the end.

- A *city block* consists of *buildings* which consist of a *roof* and a number of *stocks*, which consist of *walls*, which consist of *wall pieces* and *windows*…

Shapes and Parameters

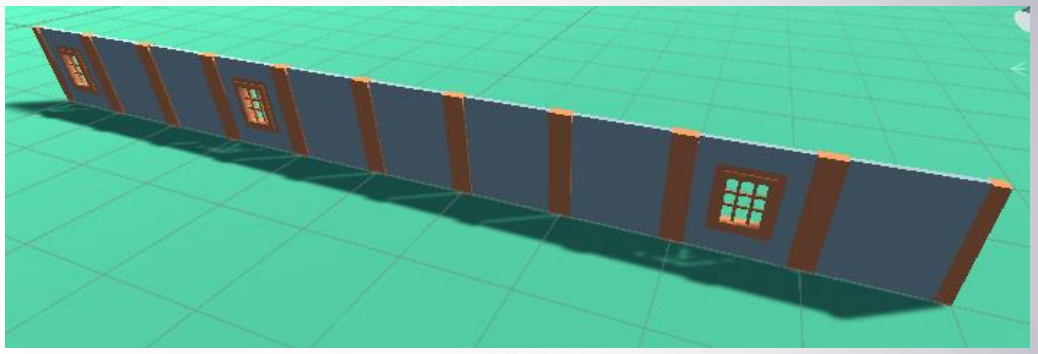
- A *shape* is a *symbol* together with *parameters*
- Typical parameters:
  - *Transform* (=position, rotation, scale)
  - *Size:* Width / height / depth
  - *Color / material etc.*

Example:

- A wooden box of dimensions 2 x 2 x 2 at position (0,4,0).

Rules

- A grammar contains *rules* that state how non-terminal symbols can be replaced.
- You can specify the *probability* of applying each rule.



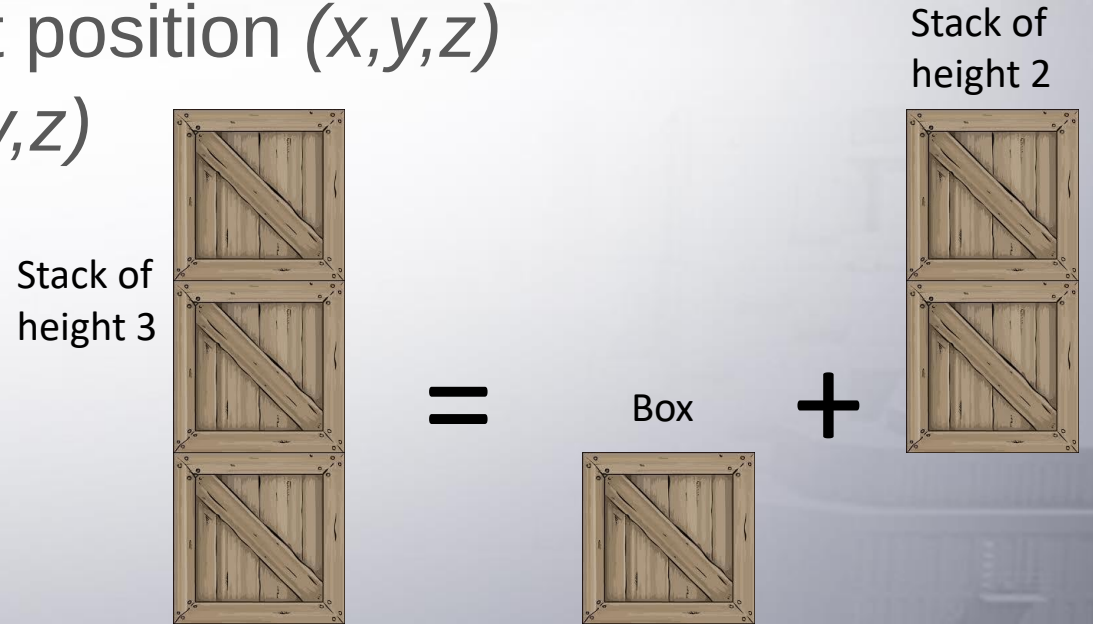
Example rules:

- A *wall* of length *n* can be replaced by *n wall segments*
- A *wall segment* can be replaced by a *window piece (25%) or* a *wall piece (75%).*
- A *stack (of boxes)* of size *n* at position (x,y,z) can be replaced by
  - a 1x1x1 *box* at position (x,y,z) *and*
  - a *stack* of size *n-1* at position (x,y+1,z).

Example Grammar

Grammar for creating a stack of boxes:

- Non-terminal symbol: S (=stack)
- Terminal symbol: B (=box)
- Shapes:
  - $S_{n,x,y,z}$ A stack of height *n* at position *(x,y,z)*
  - $B_{x,y,z}$ A box at position *(x,y,z)*
- Rules:
  - $S_{n,x,y,z} \rightarrow B_{x,y,z} + S_{n-1,x,y+1,z}$ (Only if n>1!)
  - $S_{1,x,y,z} \rightarrow B_{x,y,z}$



Why?

- Our goal is not to completely *formalize* these grammars
  - That only makes a simple concept complex!
  - We use a known tool anyway to express these grammars (C# scripts)

- Goals:
  - Understand the power of this concept!
  - Use it as a sketching & communication tool!
  - Use an implementation of it in your toolbox!

- More information:
  - An excellent lecture by Rachel Hwang: https://www.youtube.com/watch?v=t-VUpX-xVo4

Implementation

How to translate this idea to a *Unity implementation* such that we have an easy to use, easy to understand tool?

- *Terminal symbols* are *game objects* (=instantiated prefabs).
- *Non-terminal symbols* are *MonoBehaviour classes,* attached as components to game objects.
- *Parameters* are *data fields* in these classes, or in the case of transform (position etc): the *transform component* of the game object.

- *Rules* are *methods* in these classes.

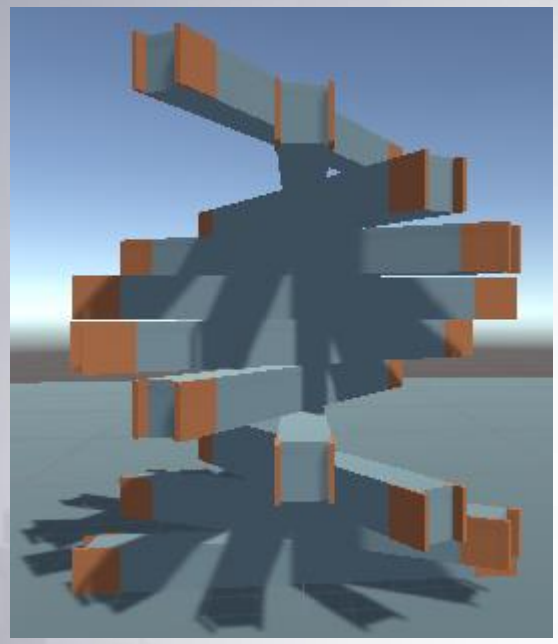
➔Let's have a look at the example grammar for building stacks (...scene…)

➔Useful code is given in the superclass *Shape!*

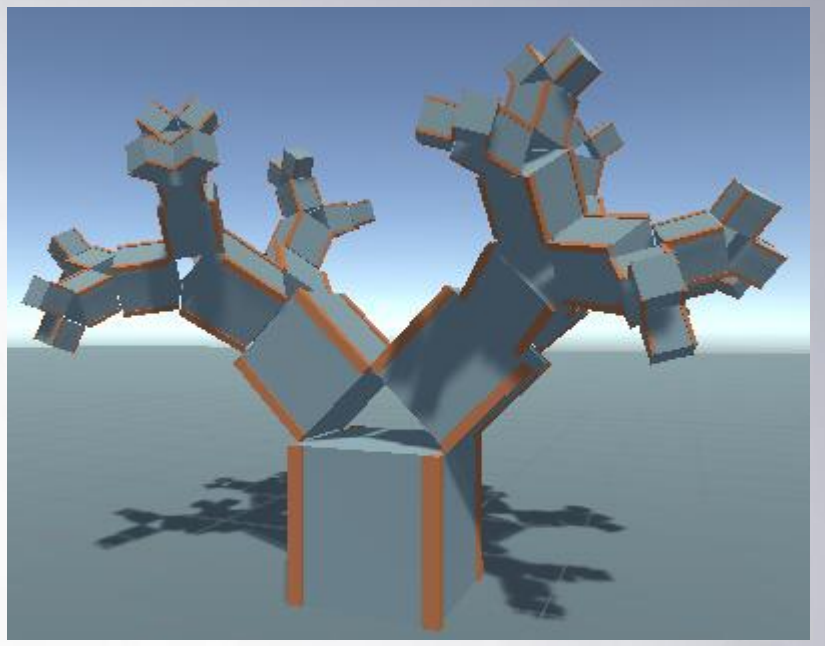
Unity Tools for Shape Grammars

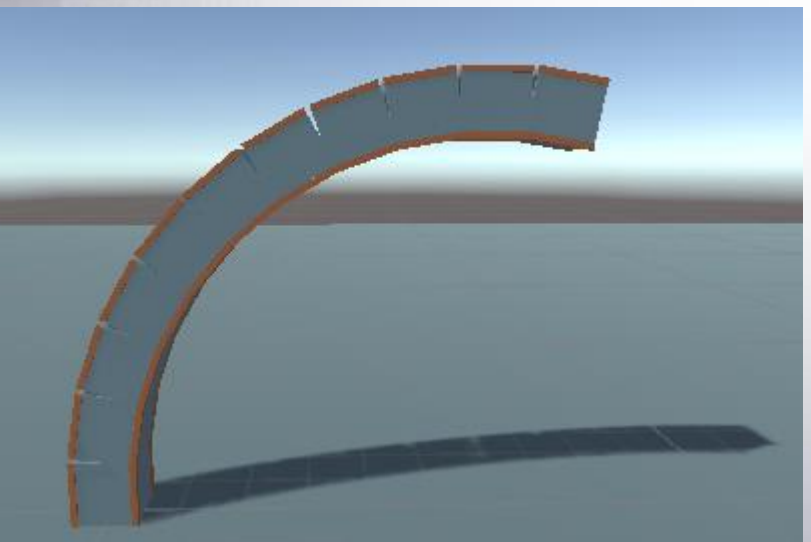
- Given scripts (in Assets):
  - *Shape*: use this as superclass for any grammar symbol, in any custom grammar
  - *BuildTrigger*: can be used to start the grammar
- Method for using this in a scene:
  - Create a game object with a component that inherits from *Shape* (such as *Stack)*, and a *BuildTrigger* component, and possibly a *RandomGenerator* component
  - Enter play mode, and press the key shown in the *BuildTrigger* inspector to start the grammar
- Creating your own grammar:
  - Create at least one symbol class that inherits from *Shape*
  - This class must implement the *Execute* method, where the grammar rules are applied
  - In this *Execute* method, you typically call the *CreateSymbol* and *SpawnPrefab* methods.
  - Typically, your own symbol class also has parameters such as *width* and *height,* and you add an *Initialize* method to assign these values.
- → It's best to have a look at an example: open the *Stack* script

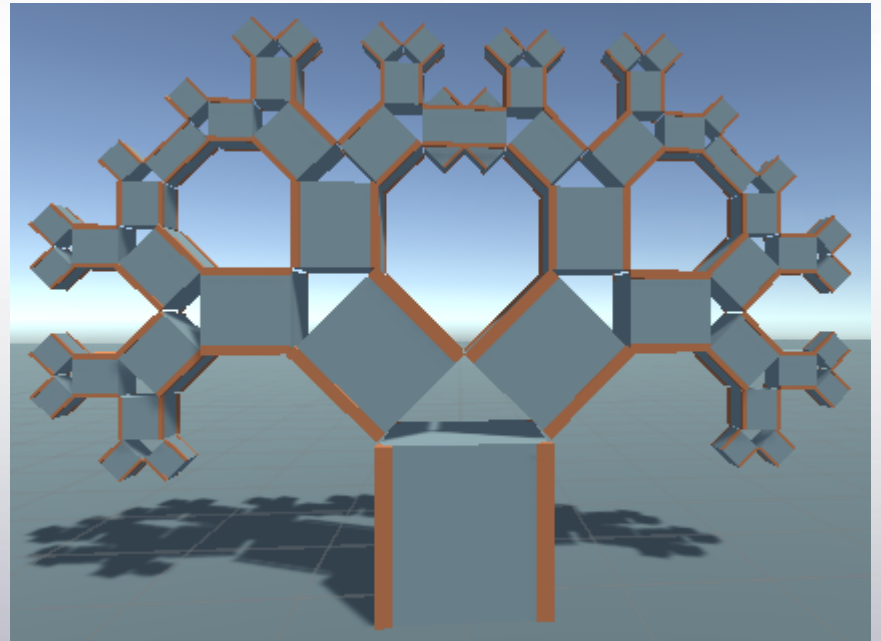
Exercise

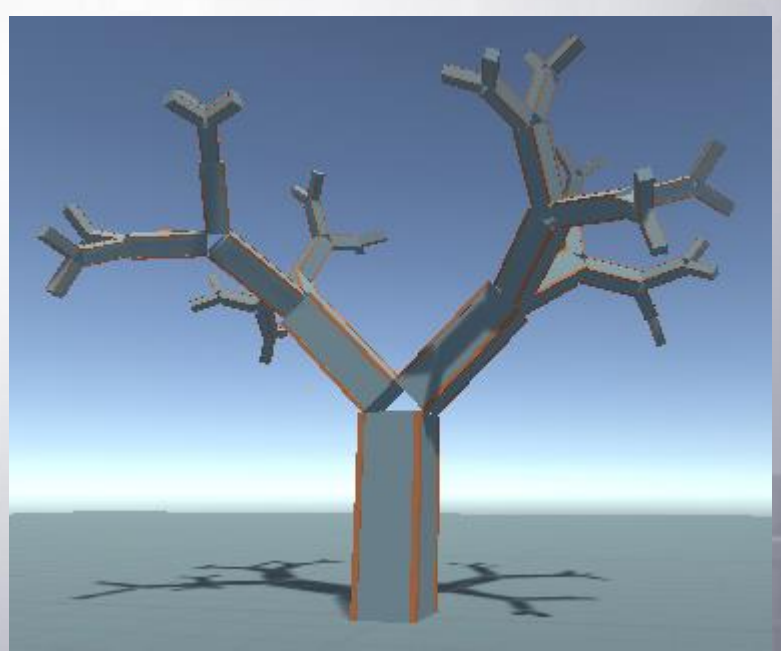


- Open Stack.cs, and try different position / scale / rotation values for the *box* and *child stack*.
- (Some suggestions are given – just uncomment.)
- *What kind of shapes can you create…?*



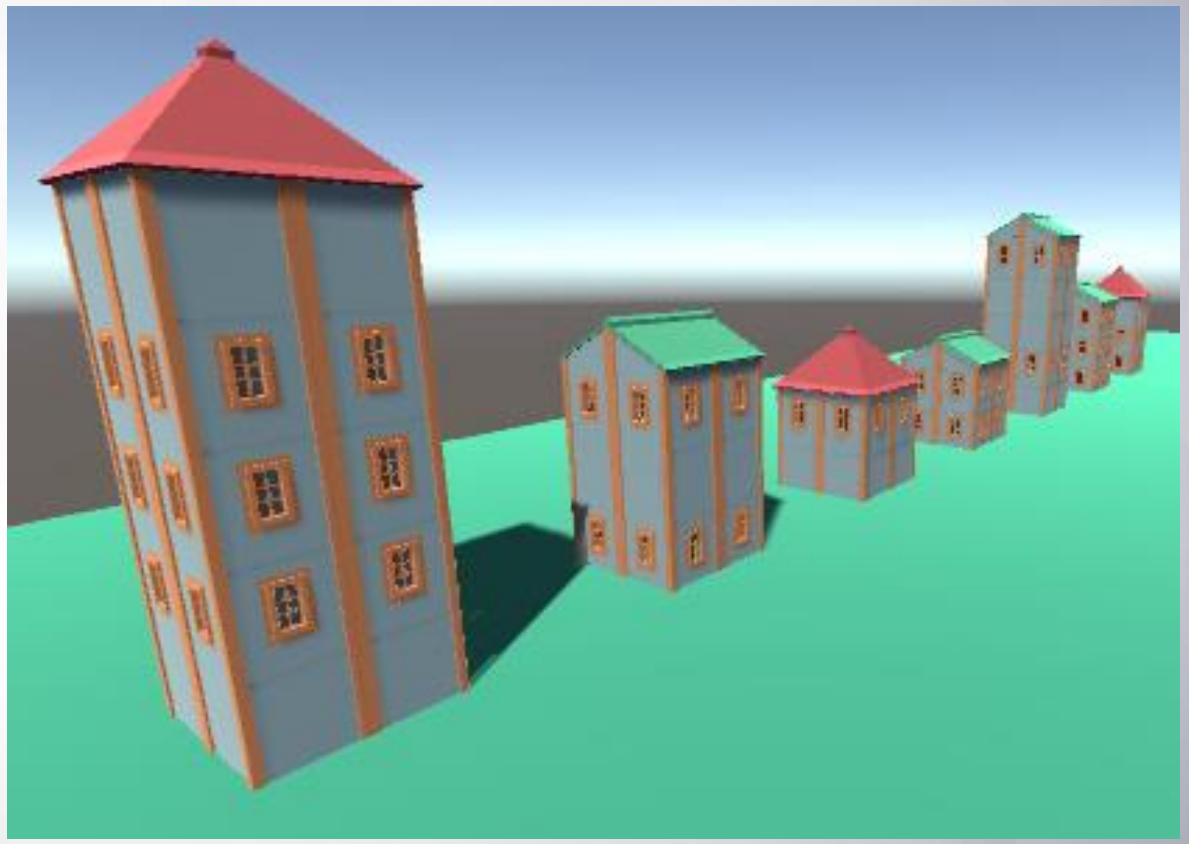






Example: Simple Buildings

- The *SimpleBuilding* grammar just has one symbol
- It is very similar to the *Stack* grammar, except that:
  - For the stocks, it chooses a random prefab from a given list
  - When the stack is done, it tops it off with a random roof prefab



Issues

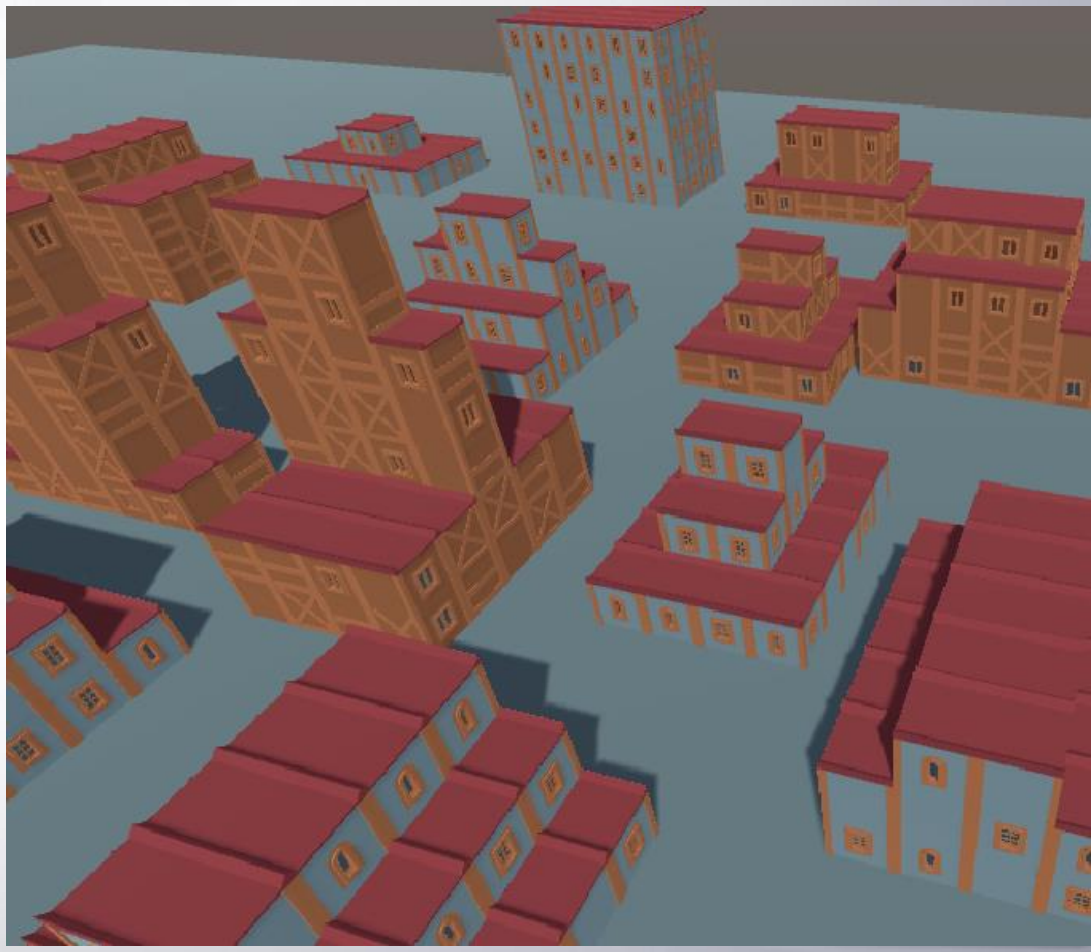
- Hard to get a “wide range of shapes” this way, without creating a ton of prefabs
- All buildings have the same width & depth (unless you create another large batch of prefabs)
- The shapes are always block-like

- Let’s see if we can come up with more interesting grammars, that create more variety

Example: Flat Roof Buildings

Grammar (sketch):

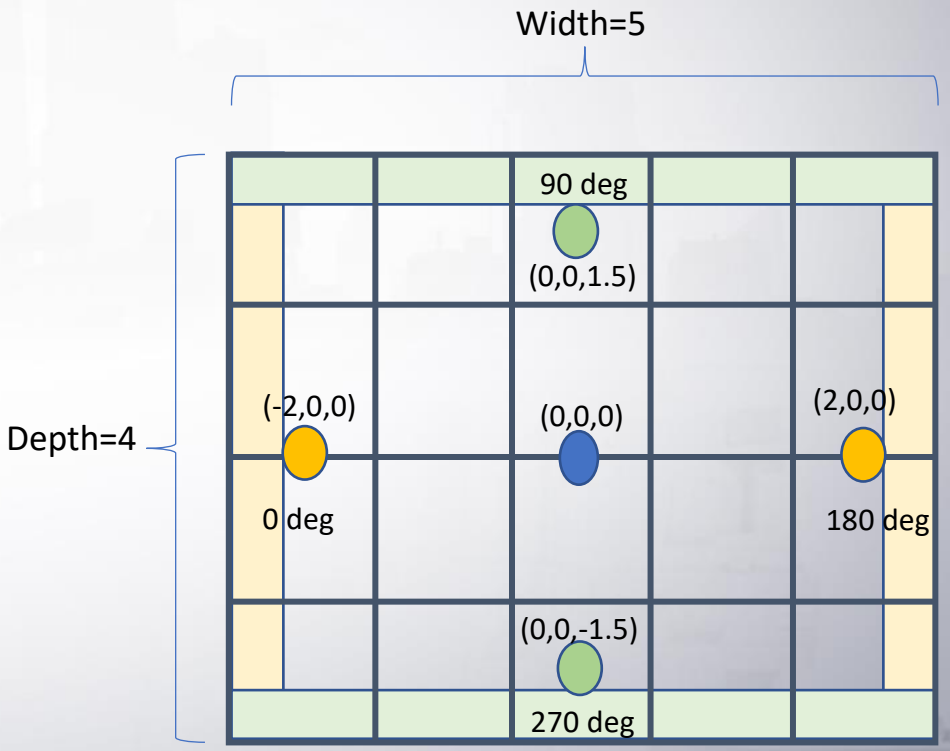
- $Stock_{h,w,d}$ → 2 x $Wall_w$ + 2 x $Wall_d$ + $Stock_{h+1,w,d}$
- $Stock_{h,w,d}$ → 2 x $Wall_w$ + 2 x $Wall_d$ + $Roof_{h+1,w,d}$
- $Roof_{h,w,d}$ → 2 x $RoofStrip_d$ + $Roof_{h,w-2,d}$
- $Roof_{h,w,d}$ → 2 x $RoofStrip_w$ + $Roof_{h,w,d-2}$
- $Roof_{h,w,d}$ → 2 x $RoofStrip_d$ + $Stock_{h,w-2,d}$
- $Roof_{h,w,d}$ → 2 x $RoofStrip_w$ + $Stock_{h,w,d-2}$
- $RoofStrip_n$ → n x RoofPiece
- $Wall_n$ → n x WallSegment
- WallSegment →WindowPiece
- WallSegment → WallPiece



Math Explanation

- In general: use *pen and paper* and some example values to understand / come up with such formulas! (That's how I do it too!)
- For this once, I'll explain the math of *SimpleStock.cs*:

```
protected override void Execute() {
    // Create four walls:
    for (int i = 0; i<4; i++) {
        Vector3 localPosition = new Vector3();
        switch (i) {
            case 0:
                localPosition = new Vector3(-(Width-1)*0.5f, 0, 0); // left
                break;
            case 1:
                localPosition = new Vector3(0, 0, (Depth-1)*0.5f); // back
                break;
            case 2:
                localPosition = new Vector3((Width-1)*0.5f, 0, 0); // right
                break;
            case 3:
                localPosition = new Vector3(0, 0, -(Depth-1)*0.5f); // front
                break;
        }
        SimpleRow newRow = CreateSymbol<SimpleRow>("wall", localPosition, Quaternion.Euler(0, i*90, 0));
        newRow.Initialize(i%2==1 ? Width : Depth,wallStyle);
        newRow.Generate();
    }
```



Issues

- Random/uncontrollable: Building is different every time we generate it
  → Use a *pseudo random generator* with a fixed *seed*

- Window pattern is chaotic
  → Use a pattern as *global parameter*

- Either every building looks similar (all wood, same windows), or buildings are a hodgepodge of styles
  → Use a *style* (set of prefabs) as *global parameter*

- Not customizable: e.g. putting a door at the ground floor
  → Use a *custom inspector* with *buttons*

Example: Facade

- To show how we can solve those issues, we first go to a simpler grammar: Façade
  - A *Façade* consists of a *Row* (of wall parts), with a smaller *Façade* on top

- We will show how to make this grammar more controllable/customizable by:
  - Adding a seeded pseudorandom generator
  - Adding a global parameters component, which determines the style and pattern
  - Adding a basic editor script for *Row: RowEditor*

- Most of this is given in the handout, you only need to add a few aspects yourself

Pseudo Random Generators

- Random generators:
  - Random in the sense of “totally unpredictable without pre-knowledge”
  - Not actually random in practice!
  - Every random generator initialized with the same *seed* returns the same sequence of numbers!
- Use a random generator from System.Random (not from Unity!)
- Initialize it with the seed: myRandomGenerator = new Random(seed)
- Your task: implement this in the *RandomGenerator* script.

```
> Random rand = new Random(0);
> rand.Next(1000)
726
> rand.Next(1000)
817
> rand.Next(1000)
768
> rand.Next(1000)
558
> rand=new Random(0);
> rand.Next(1000)
726
> rand.Next(1000)
817
> rand.Next(1000)
768
> rand.Next(1000)
558
> rand=new Random(1);
> rand.Next(1000)
248
> rand.Next(1000)
110
```

Global Parameters

- Add a component to the root game object (e.g. *FacadeParameters)* that contains all global parameters

- The given *Shape* super class takes care of passing a reference to the root game object around!
- You can get these parameters from any child object using *Root.GetComponent<FacadeParameters>()*

- Example parameters:
  - Prefabs (e.g. wood walls vs stone walls / round windows vs square windows)
  - Probabilities (e.g. the probability to continue with a stock after a roof)
  - Pattern arrays (to get regular stocks)

- See the Façade grammar for an example

Demo

- Using global parameters & random seeds

- Customizing the result

  → See RowEditor.cs for an example of how to create a custom inspector with buttons!

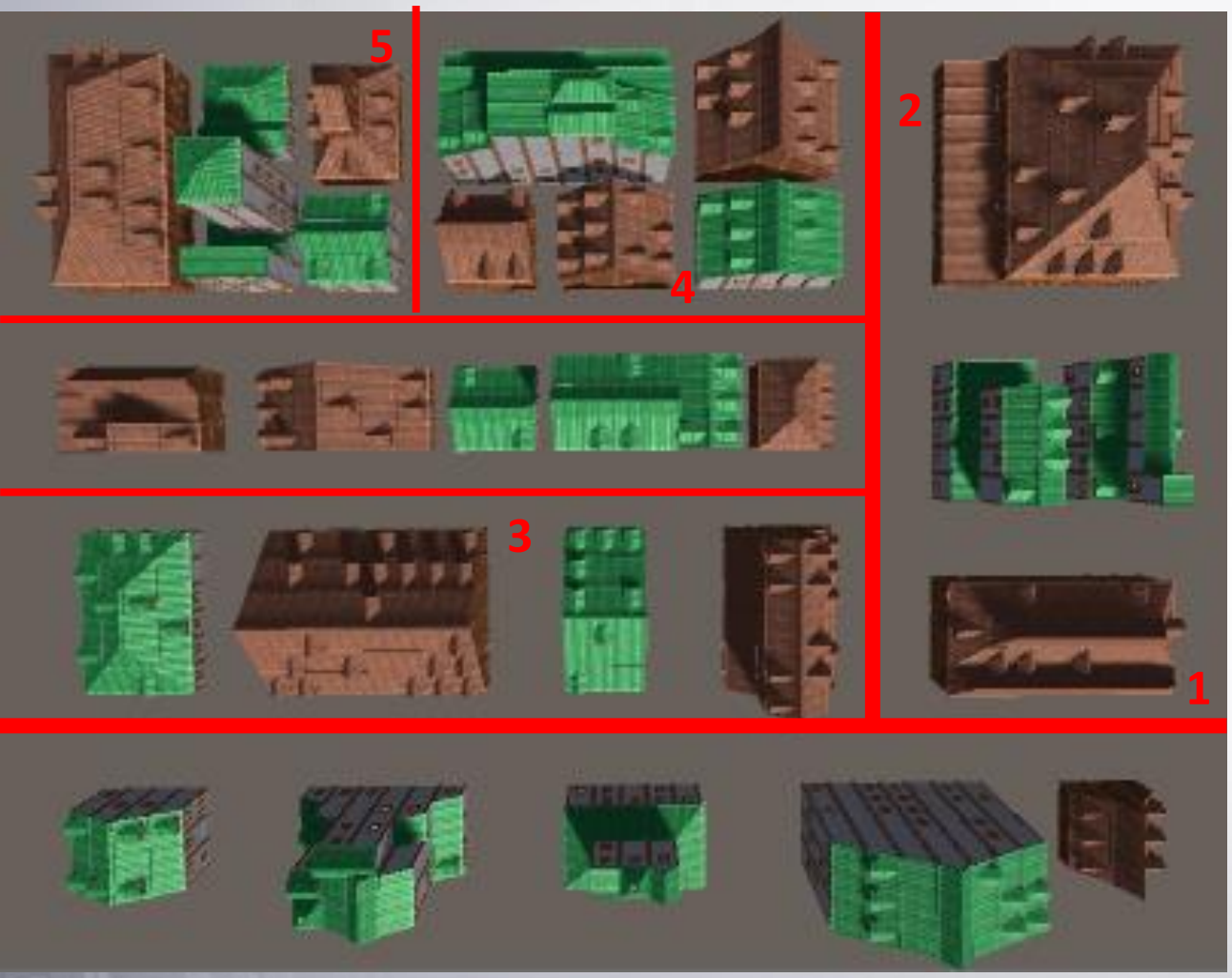
Grammar Example: Creating a City Shape

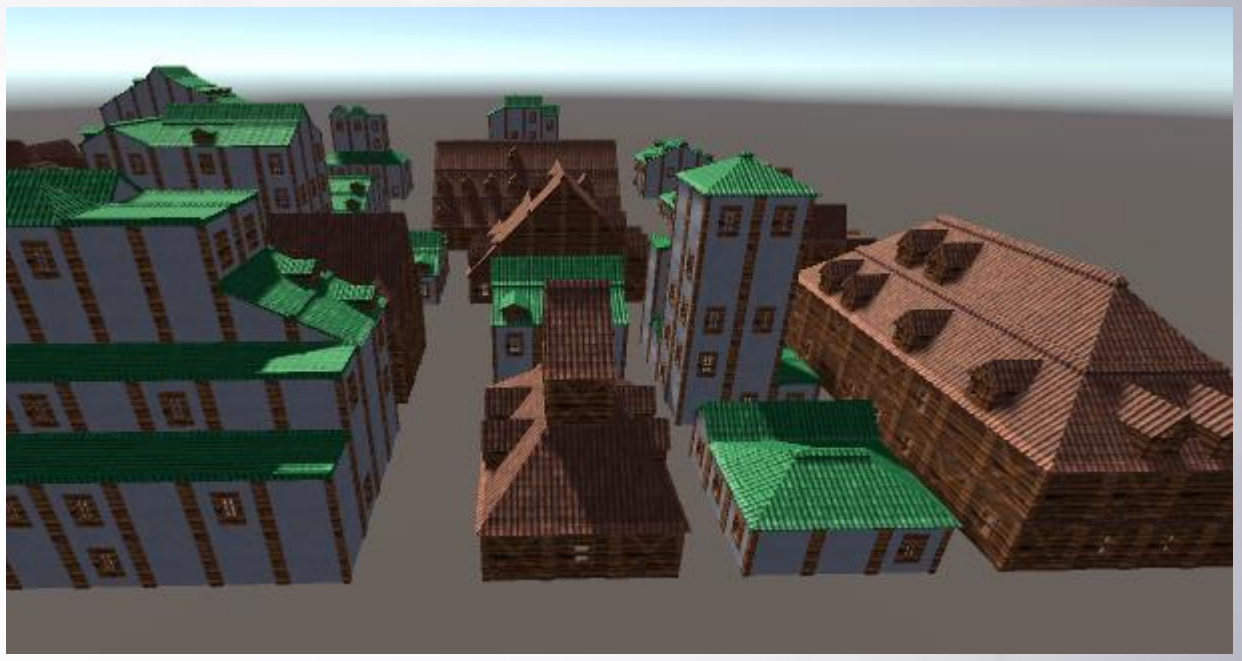
- The given *GridCity* script gives a boring and regular city layout, with long roads
- You can use a grammar inspired by “Binary Space Partitioning” to get more interesting shapes:
  - Start with a rectangle with certain width & depth
  - For each rectangle:
    - if it’s large enough, randomly split it into two rectangles using a road, either in east-west direction or north-south direction
    - If it’s small (in both dimensions), place a building

- As a shape grammar:
  - $\text{Rect}_{w,d}$ → $\text{Rect}_{w1,d}$ + $\text{N/S-Road}_{rw,d}$ + $\text{Rect}_{w-w1-rw,d}$ (if w is large enough to split)
  - $\text{Rect}_{w,d}$ → $\text{Rect}_{w,d1}$ + $\text{E/W-Road}_{w,rw}$ + $\text{Rect}_{w,d-d1-rw}$ (if d is large enough to split)
  - $\text{Rect}_{w,d}$ → $\text{Building}_{w,d}$ (if both w and d are small)

  - (rw = road width. w1 and d1 are the new width / depth, chosen randomly but such that both rectangles have positive width and depth)

Result





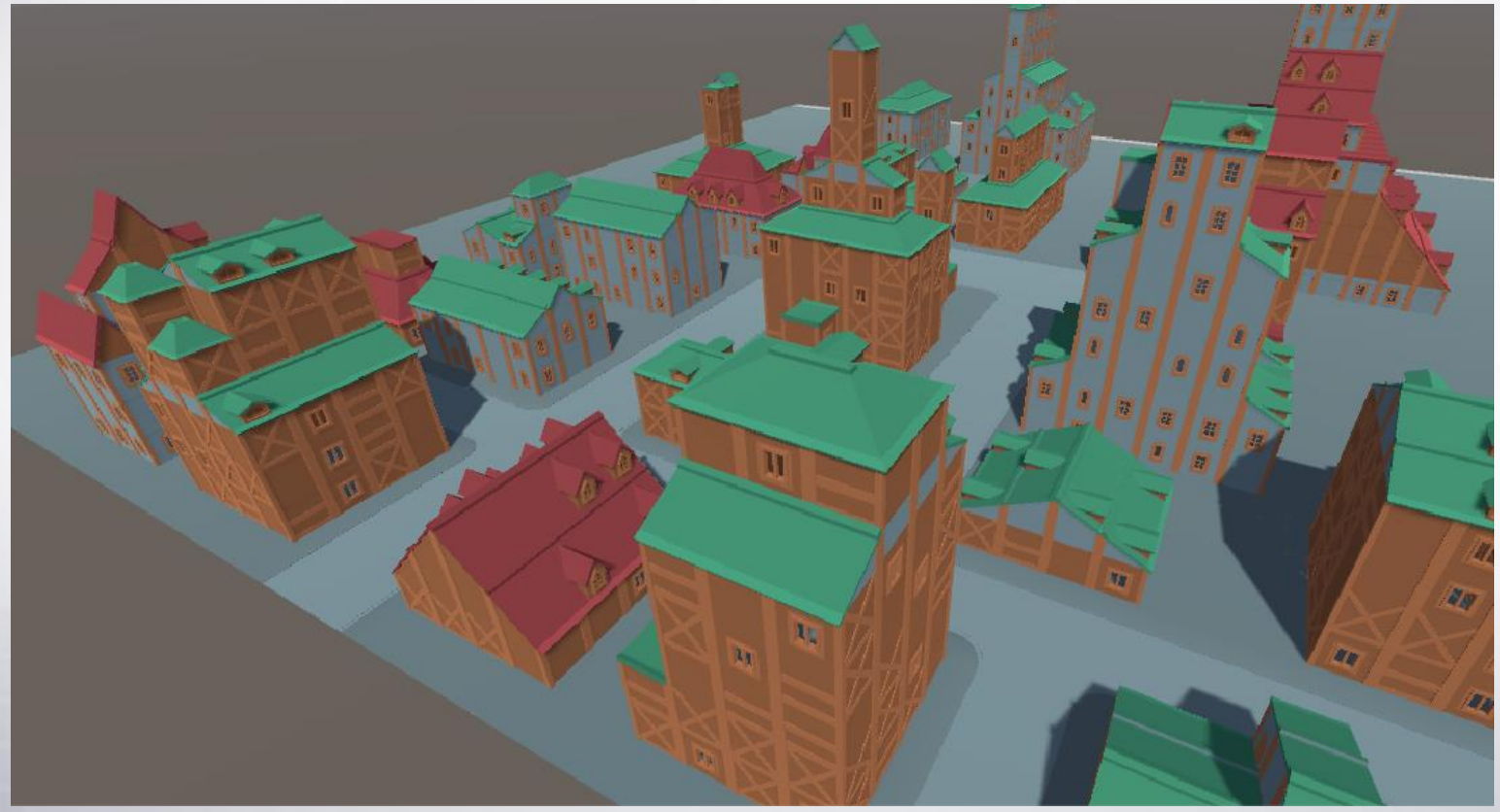
Extra trick applied here:

- add a *road width* parameter,
- randomly make it smaller for each split

→ Result: short streets are narrower (=realistic)

Advanced Grammar Example

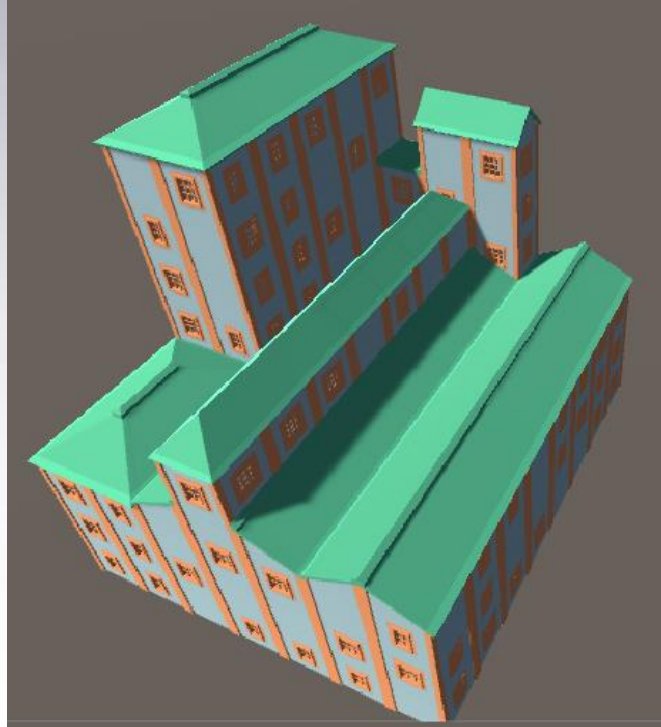
- Creating a Kenney town variant in real time…

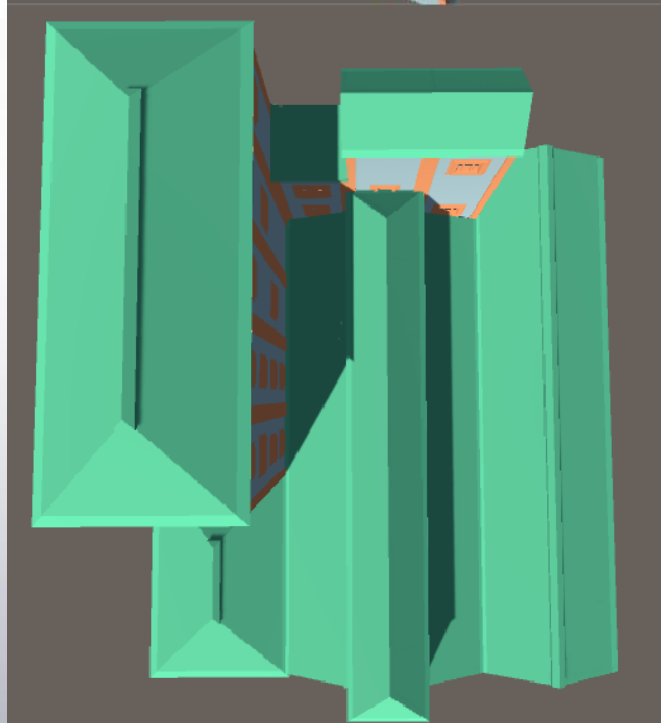


Grammar Extensions

Additions to the grammar:

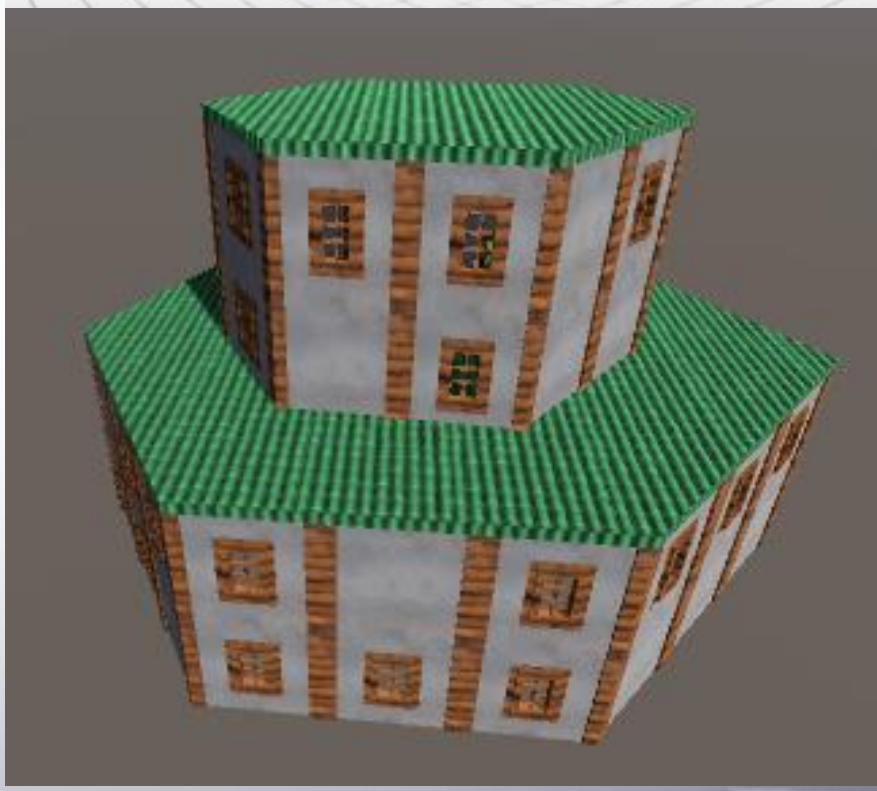
- Four roof types: flat, side slopes, front/back slopes, four slopes
- Additional symbol: *Rectangle*
  - …can be replaced by a *Stock*
  - …can be replaced by a *Roof*
  - …can be replaced by *two smaller Rectangles*!
- …but anyway, don’t copy my grammar, *get creative!*





Tips - Shapes

- We didn't use *scaling* or *rotation,* so all our building blocks fit into a 3D "voxel grid".
  - This is not a requirement – get creative!
  - You can use any *polygon (=list of Vectors)* as a ground floor, and generate walls for it
    - This involves a little bit of scaling for the walls + some vector math from Physics Programming (atan2, normal, …)
    - How to generate a roof…? → a topic for the third scripting lecture!
    - (demo)
    - Note: the editor tools used here (gizmo's) will be explained in next week's lecture!



Tips - Shapes

- Using *scaling* and *rotation* you can also generate completely different shapes, like say *trees.*

- Be careful with extreme scaling, since this changes the texture scale (texel density) as well!

Tree examples: generated by *Joey Poortman*, using these tools + custom models

Tips – Input Textures

- You can use some global texture data (e.g. a terrain texture / noise pattern) as *input*
- For example:
  - City center: high buildings
  - Suburbs: low buildings (+lots of space between buildings)
- When creating a building at a certain position, sample the texture to determine its height
- For an example of this idea (using Perlin noise), see e.g. https://www.youtube.com/watch?v=xkuniXI3SEE
- If you use a manually created texture, you have some control over the layout of the city (tradeoff: realistic results vs input size)

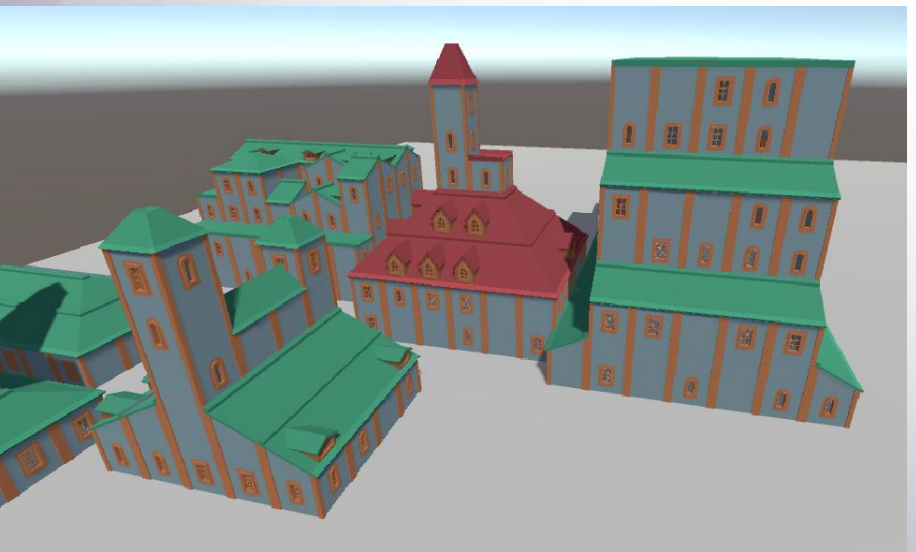
Level of Detail (LOD)

- Let’s look at Novigrad: https://youtu.be/34SvvSIkRjk?t=210

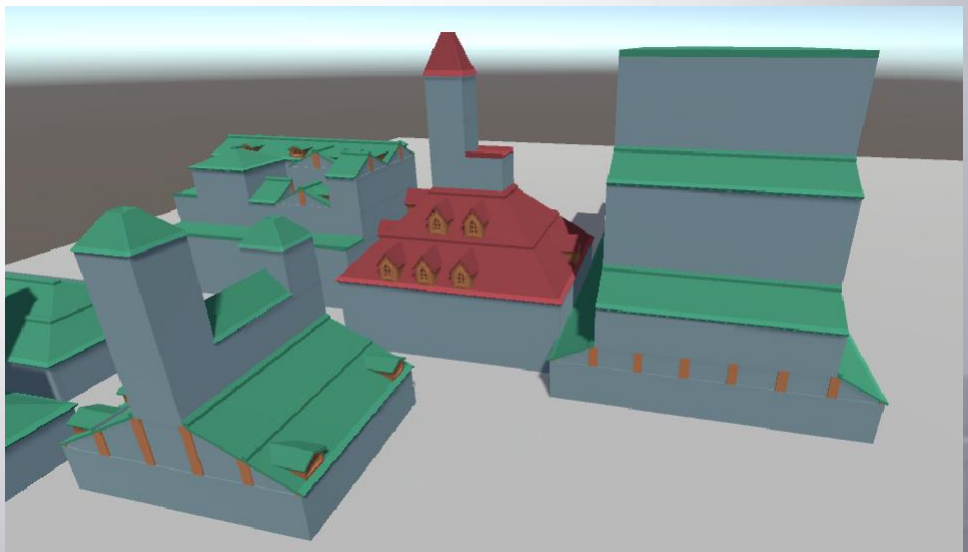
Level of Detail

- In almost any large scene (game / movie), varying *level of detail (LOD)* values are used to manage the number of triangles on screen:
  - Far away meshes: low detail / few triangles
  - Close by meshes: fine detail / many triangles

- Shape Grammars work excellently for this! Examples:
  - Tree: don’t spawn the finest branches
  - Building: for a stock, spawn just a cube in the right color



(demo)



Scale & LOD

- If you want to show a huge city, and have detailed 3D models, you need to manage LOD

- *Advanced technique:* You can do this dynamically:
  - Spawn a more detailed version of a model when the camera gets closer
  - Keeping track of random seeds is essential!

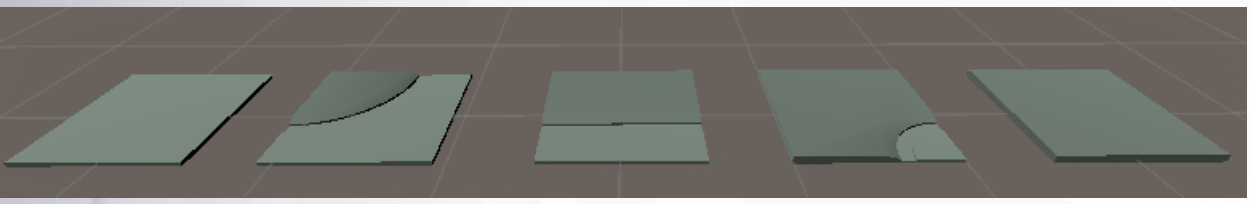
- Unity can also do it for you, if you define LOD groups:
  - https://docs.unity3d.com/Manual/LevelOfDetail.html

Grid Based Methods

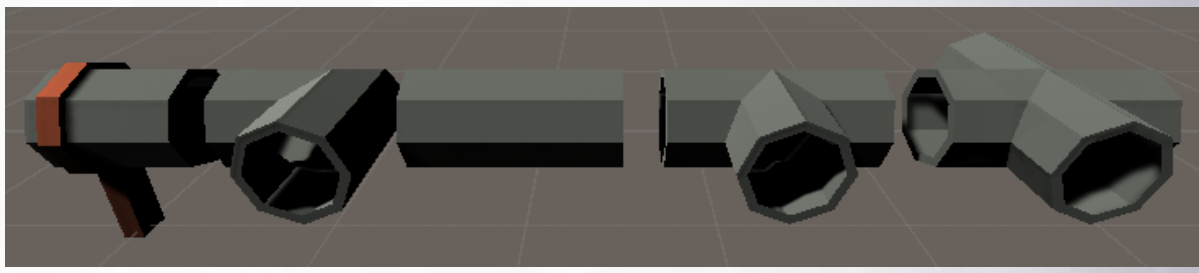
Modular Roads / Pipes / Walls / …

- Lets have a look at some more modular assets from Kenney:

Road pieces:



Pipe pieces:

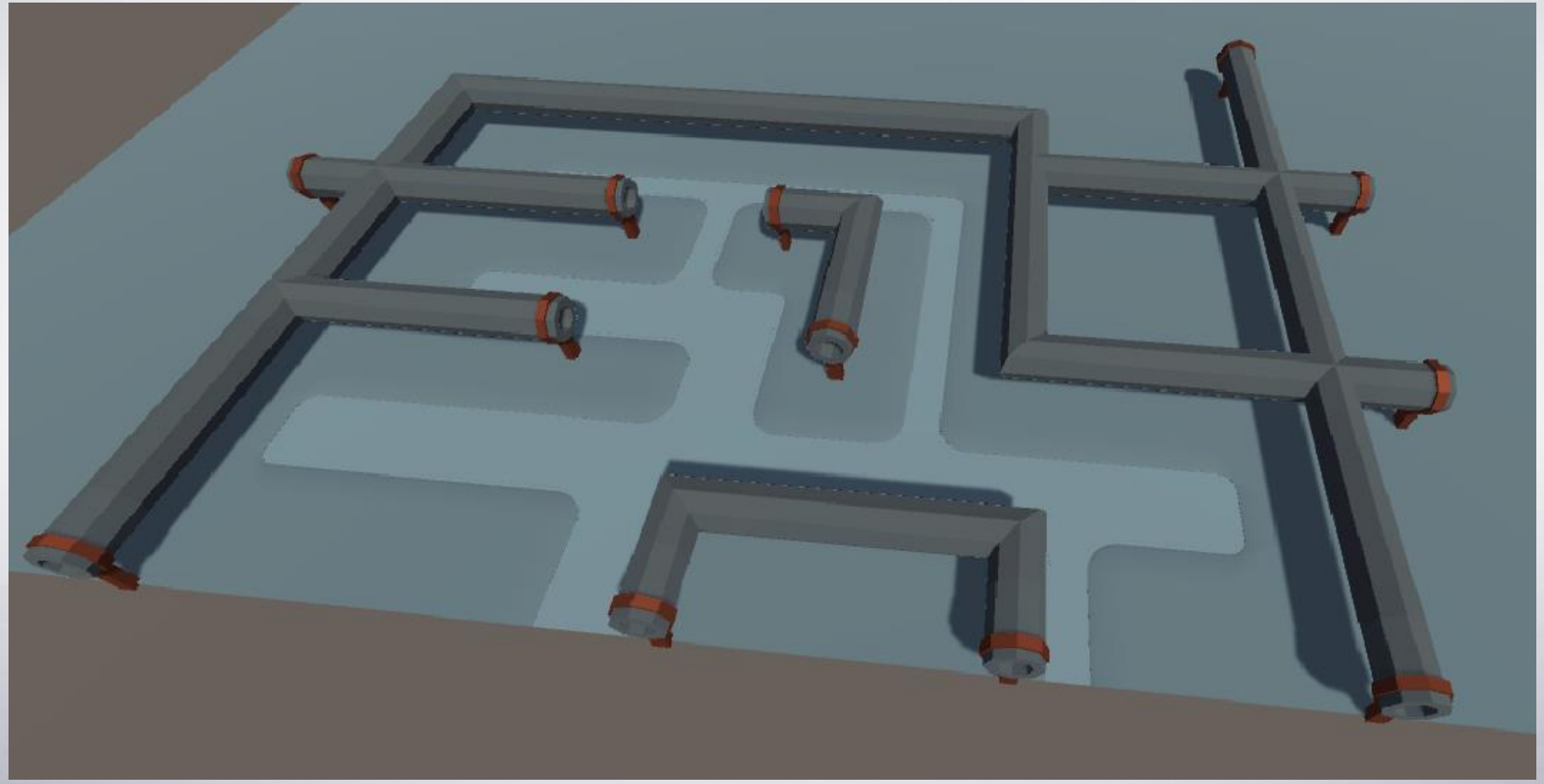


- How can we use these?

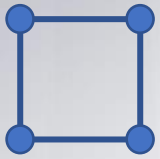
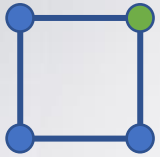
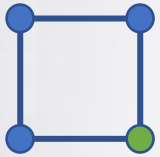
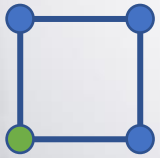
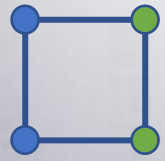
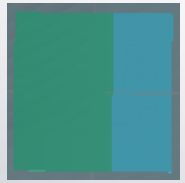
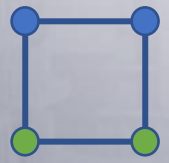
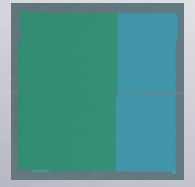
- …it would be great if we can just say *where* we want the road to be, and then have an algorithm that connects the parts correctly!

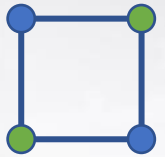
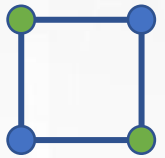
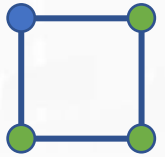
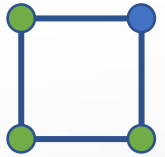
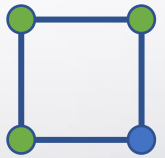
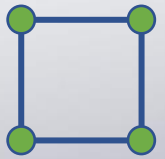
→ *Marching squares* (…and related methods)

Demo

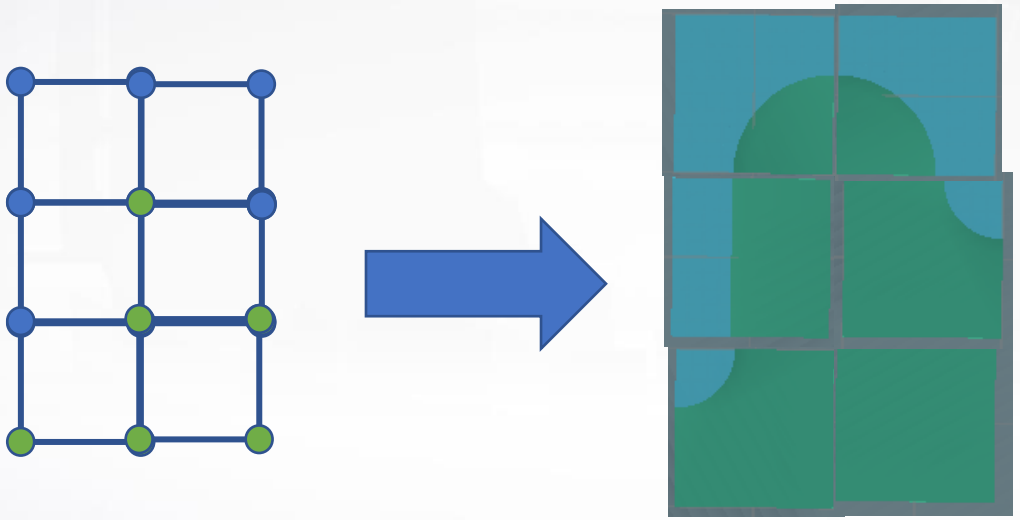


Main Idea

Grid corners:	Bitmask:	Prefab:	Rotation:
	0000		0º
	0001		180º
	0010		90º
	0100		0º
	0011		180º
	0110		90º

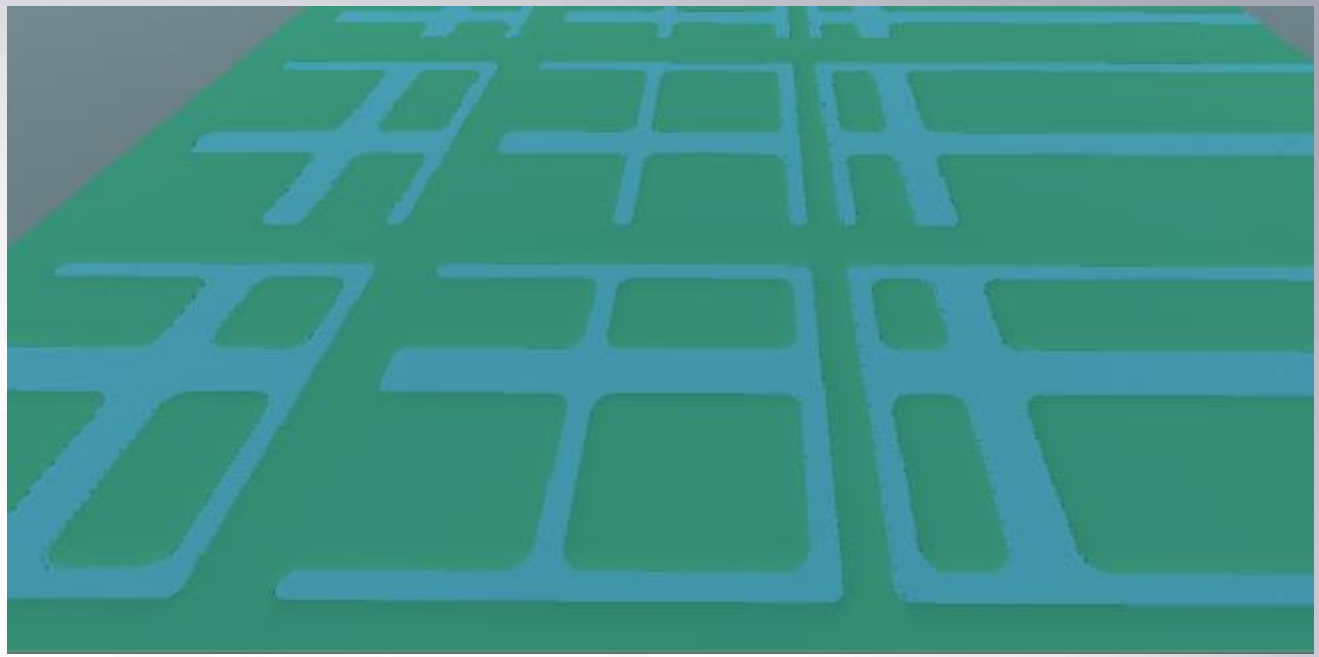
Grid corners:	Bitmask:	Prefab:	Rotation:
	0101	???	0º
	1010	???	90º
	0111		90º
	1110		0º
	1101		270º
	1111		0º

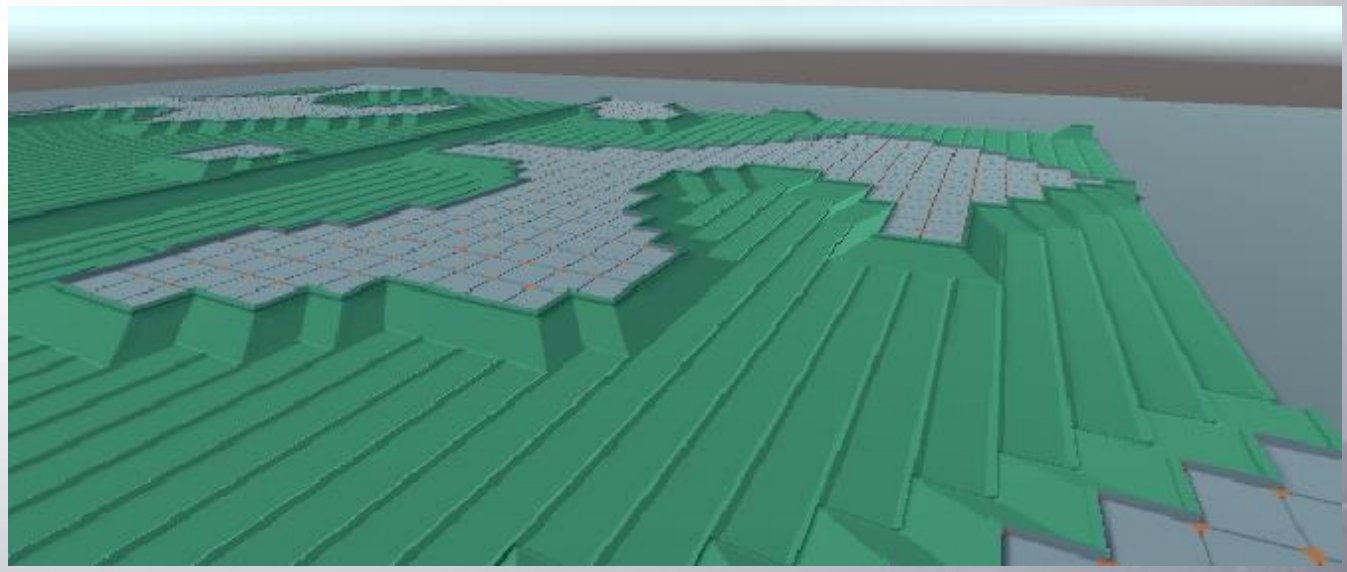
Result



Exercise

- Open MarchingSquares.cs and ValueGrid.cs
- In the InitializeGrid method, try out different ways of initializing the grid
  - Some examples are given (e.g. Perlin Noise)
  - What else can you create…?
- Try dragging in different prefabs, such as roof parts
  - What can you create…?
  - Do you need to adjust the base rotations?





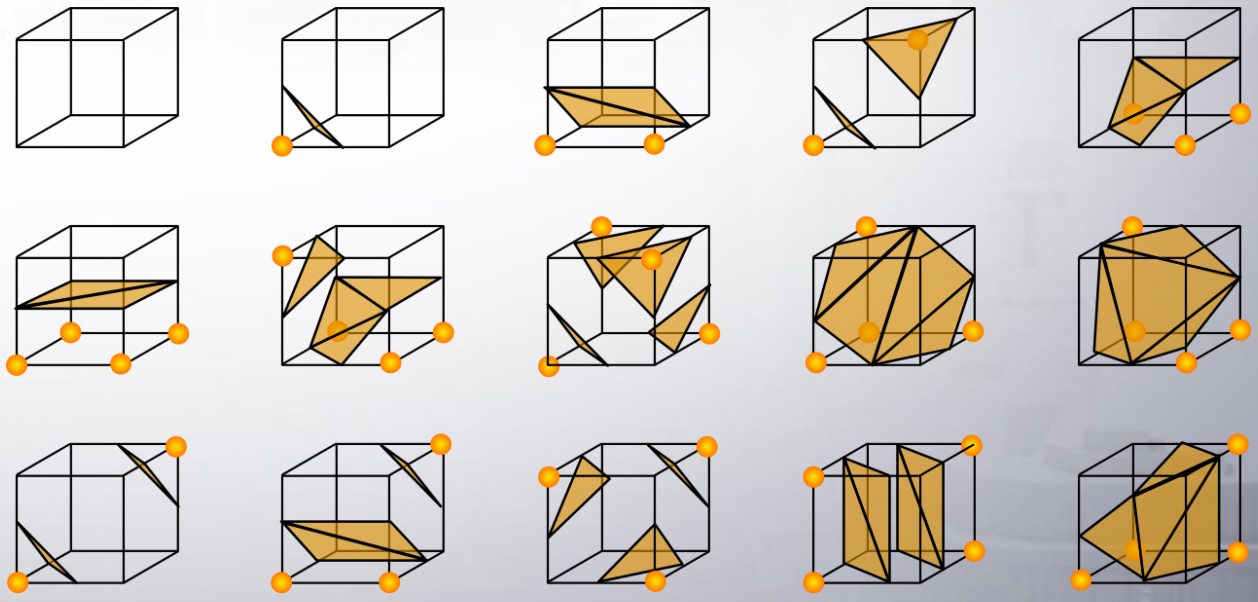
Marching Squares - More Information

- https://en.wikipedia.org/wiki/Marching_squares
- https://www.youtube.com/watch?v=yOgIncKp0BE

Marching Cubes

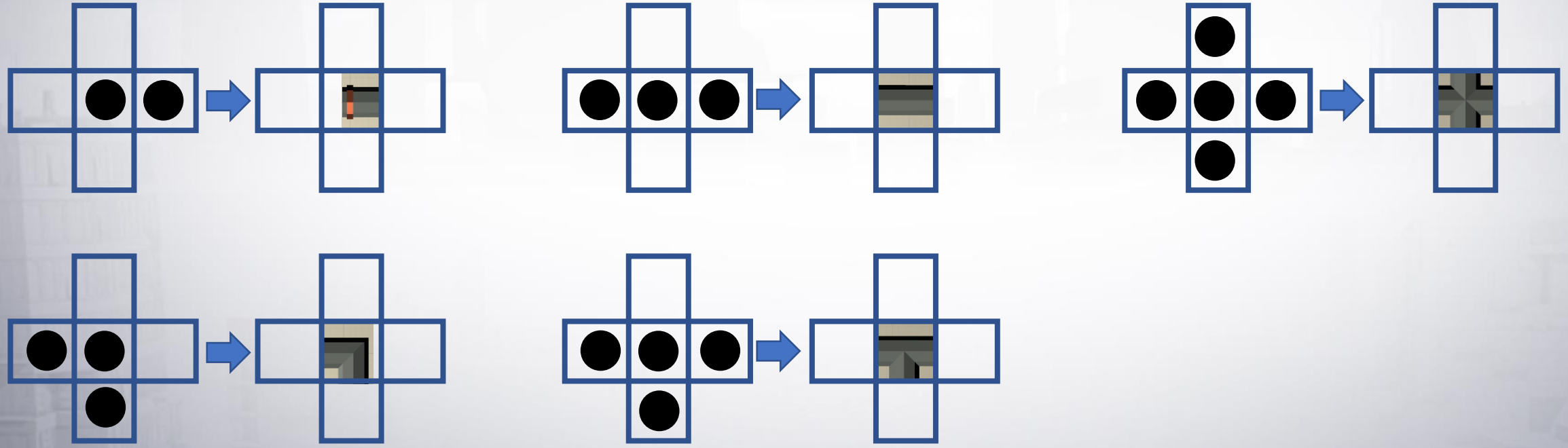
- *Marching cubes* is the same as marching squares, but in 3 dimensions
  - You need 15 different meshes
  - Just like the one case in marching squares, some cases are ambiguous – multiple meshes fit
  - 8 corners, so $2^8 = 256$ cases! (instead of $2^4 = 16$)
  - That's a big table to fill by hand… → luckily other people have done that already

- More information:
  - https://en.wikipedia.org/wiki/Marching_cubes
  - http://paulbourke.net/geometry/polygonise/



Variant

- For the *pipe assets*, we can do something similar:
  - don’t consider *grid corners*, but *adjacent grid cells*
  - Small modification of the given code



Conclusion

Summary

- Powerful techniques for procedural generation using meshes:
  - Shape Grammars
  - Grid based techniques: marching squares / cubes

- The given scripts and grammars help you to get started
- Adapt them to your needs!
- Create your own shape grammars!

- Study the resources given here (and more…)

Tips

- When creating shape grammars:
  - Test every step!
  - …so start with the basic building blocks (“wall” first, “city block” last)
  - Use pen & paper to figure out the right values! (position vectors, etc)

- Technique & algorithms only get you so far…: study your inspiration! (Real world building styles)

Next Lectures (Scripting)

- Next week: Unity editor tooling
- After that: creating & modifying meshes by code

More (General) Resources

- http://pcgbook.com/
- http://pcg.wikidot.com/
- Survey of PCG techniques (Kate Compton): https://www.youtube.com/watch?v=WumyfLEa6bU
- www.citygen.net/files/Procedural_City_Generation_Survey.pdf