Welcome to Polygon Precinct!

Exploring Structure with Generative Modeling Techniques

Joanna Gerr

Massachusetts Institute of Technology

May 13, 2019

Abstract

In recent years, procedural generation has grown in popularity as a means of algorithmically generating large amounts of content for various computer graphic (CG) and video game applications. One of the most practical uses for procedural generation is to model large environments filled with repetitive structures—like cities, which features as environments in many video games and other CG-reliant media. While procedural generation for city-like structures that resemble cities superficially are not hard to come by, it is difficult to find generative cities that take into account facets of structural planning and social engineering to represent the intricacies of urban development. To address this, I have looked into existing methods of generating cities and the structures within them at both the 2D and 3D scales, compared them against real-world structural and social concerns, and improved upon representations for cities in ways that begin to account for the depth of urban life.

I. Introduction

In the modern era, urbanity is inescapable. The city is a sprawling, massive construct, come into being over centuries of hard labor, both inside and outside—and naturally, such a continual process is subject to changes in its trajectory over time. But what are the factors that shape this trajectory? What has led to cities becoming the structures we recognize them as today—where steel-and-glass skyscrapers tower across the street from rows of packed-together townhouses?

There is no singular answer to that question. However, with the literature of urban scientists and architects like Jane Jacobs and Matthew Frederick, I have been investigating the factors that contribute to making the city what it is—both from a structural and a social standpoint—and then, attempting to replicate cities based on an assemblage of these factors.

Traditional methods of procedural generation focus on algorithms as a means to an end. Often, procedural generation is to model vast stretches terrain or terrain-like patterns: for instance, in generating 2D textures to simulate water, stone, and wood; or instantiating different types of trees for the forests of an RPG; or for the layout of a dungeon. Dungeon design is one of the oldest uses for procedural generation, having been applied to pen-and-paper Dungeons and Dragons campaigns even before CG interfaces could visualize the dungeons properly. Likening pathway creation to designing mazes, some of the first attempts at procedural pathways for dungeons focused assembling perfect mazes: "2D, normal, orthogonal, ...which simply means that the maze is rectangular, with all passages intersecting at right angles, and that there are no loops or inaccessible areas in the maze[5]." From that baseline, more paths would be added and removed to make some areas more sparse with fewer dead ends, and create cycles that allowed for easier navigation.

When applying these traditional methods to a city, we find that the resulting structure may look city-esque and be navigable as a city should be, but lacks the underlying rationale that drives real cities to manifest as they do. In the following sections, I will look at 2D and 3D representations for cities, and consider ways in

Dungeon Level: 1

Room #1:

- Door (west, 3 from north): iron, free
- Door (east, 1 from north): wooden, simple, free
- Empty

Room #2:

- Door (north, 2 from west): wooden, strong, locked
- Empty

Room #3:

- Door (east, 3 from north): wooden, simple, locked
- Door (west, 3 from north): wooden, strong, stuck
- Empty

Figure 1: *A sample of 3 rooms from Jamis Buck's Dungeon Generator[4] with default settings.*

which computer-generated models can account for a greater spectrum of structural and societal urban considerations.

II. Structure

In this first part of my research, I focused on methods of generating 3D city visualizations. The tools I used to output my own generative content were Autodesk Maya and Blender for modeling 3D assets, and the Unity game engine to compile models into larger-scale visualizations. The Maya Embedded Language (MEL) scripting language was instrumental in allowing me to procedurally generate and export different types of buildings at a high volume, but I did not want to be solely reliant upon Maya, since the access barrier to proprietary software like Maya is high for anyone who may be interested in revisiting this in the future; this being the case, instead of sticking solely to Maya, I also used Blender for other modeling and animation tasks, as it is open-source where Maya is proprietary. I chose to use Unity because it is a free, C-based game engine with strong 3D rendering support, and its more robust coding capabilities would allow me greater flexibility when scripting algorithms to assemble the cities.

As I would be generating large numbers of buildings at any one time, all representations of buildings I assembled are made of as few polygons as possible to represent the desired geometry; this artistic style of economizing polygons is called low-poly modeling. My goal was not to model the most building-like building, but rather, assemble geometries that suggested consideration of factors beyond the superficial. Hence, low-poly modeling—which is itself an abstraction of realism into shapes—matched my goals well. As Frederick says, "Geometric shapes have inherent dynamic qualities that influence our perception and experience of the built environment[6]."

i. Buildings

My first goal was to script procedural buildings of the type that would be built nowadays for a city in modern-day America. Frederick notes as tip 59 of his *101 Things I Learned in Architecture School* that "Most modern buildings employ a frame of steel or concrete columns and beams to support structural loads and transfer the building's weight to the earth... supported by the super-structure every story or two[6]." Coupling his insight with my own observations made about buildings in the modern American cities of Washington DC, New York, and Boston, I envisioned buildings that were tall and multi-story, rectangular, and with varying levels of complexity for this first batch.

To achieve this, I assembled methods in MEL that would construct buildings, taking into account a parameter to determine their complexity. In the main *makeBuilding* method from Code 1, lines 5-9 assign variables for parameters that determine the overall size (height, width, and depth) per each building with slight random variation, seen in lines 11-13. The actual polyCube object that constitutes the base for the building is created, repositioned, and regrouped into the 'building' set on lines 17-19; then, on lines 23-30, a series of additional complexity operations are performed upon the base to randomize the complexity based off the original complexity parameter passed into *makeBuilding*. Last, lines 31-35 take care of merging all instantiated objects into one building proper (rather than separated rectangular

```
global proc makeBuilding(int $complexity){
    global int $comp;
    $comp = $complexity;

    float $binLower = 1;
    float $binUpper = 2;
    float $upperBound = 10;
    float $lowerBound = 4;
    int $subdivBound = 10;

    float $bldH = rand($lowerBound, $upperBound);
    float $bldW = rand($binLower, $binUpper);
    float $bldD = rand($binLower, $binUpper);
    //...
    sets -n building;

    polyCube -w $bldW -h $bldH -d $bldD -sx $bldW_SD -sy $bldH_SD -sz $bldD_SD -ax 0 1 0 -cuv 3 -ch 1;
    move -r 0 ($bldH/2) 0 ;
    sets -in building `ls -sl`;

    string $sel[] = `ls -sl`;

    for ($each in $sel){
        int $bldSD = rand(0, $subdivBound/2);
        addWindows($bldW, $bldH, $bldD);
        addLip($bldW, $bldH, $bldD, $bldSD);
        addHat($bldW, $bldH, $bldD);
        addChimney($bldW, $bldH, $bldD);
        addAntenna($bldW, $bldH, $bldD);
    }
    select building;
    string $full[] = `ls -sl`;
    string $allObjects = stringArrayToString($full, " ");
    polyUnite -ch 1 -mergeUVSets 1 -centerPivot -name $allObjects;
    rename bld;

}
```

Code 1: *Main method makeBuilding where base structure's width, height, and depth are randomized within bounds*

prisms), convenient for exporting into Unity later. The complexity functions from lines 23-

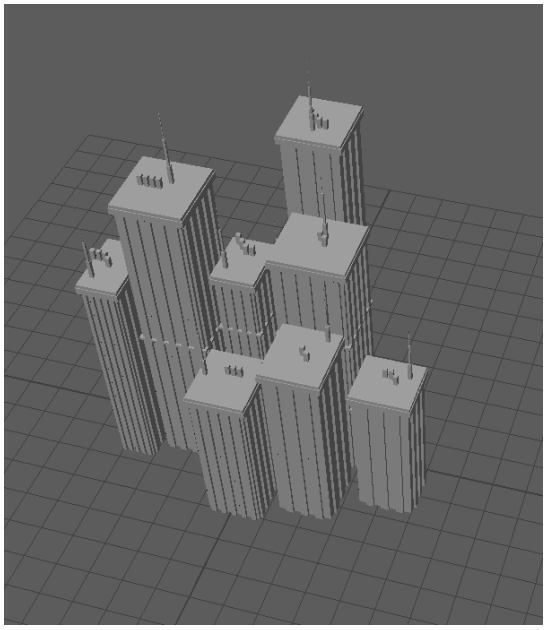

Figure 2: *First pass at abstractly representing complex buildings*

30 refer to a series of self-describing functions that construct more low-poly rectangular details that increase the perceived complexity of a building through additional features. Possible features include those found on actual buildings, such as: the number (and size) of windows, the presence (and height) of antennae and chimneys, and the appearance of a roof's edge and other lips at intervals along the building exterior. Each of these features was selected as representing an integral component of a building; windows serve to light buildings naturally during the day, and allow them to serve as light sources at night; antennae function as beacons for telecommunications happening within and around buildings; and lips serve to remind of the steel reinforcements that form the super-structure supporting buildings at multi-story intervals. Code 2 displays *addAntenna* as an example of how the method works; in lines 4-5, the method uses the given complexity to randomly determine whether the current building will have an antenna. If it

```
1  proc addAntenna(float $bldW, float
       $bldH, float $bldD){
2      global int $comp;
3      \\...
4      int $isComplex = rand(0, $comp+8);
5      if ($isComplex < 8){
6          $dim = 0.1;
7          float $xPos = rand($dim,
               $bldW-$dim) - ($bldW/2);
8          float $zPos = rand($dim,
               $bldD-$dim) - ($bldD/2);
9          float $yPos = $bldH;
10         int $numLev = rand(3, 4);
11         for ($i = 0; $i < $numLev;
               $i++){
12             polyCube -w $dim -h 1.5 -d
                   $dim -sx 1 -sy 1 -sz 1
                   -ax 0 1 0 -cuv 3 -ch 1;
13             move -r $xPos $yPos $zPos ;
14             sets -in building `ls -sl`;
15             $dim -= rand(0.04, 0.05);
16             \\...
```

Code 2: *Sub-method addAntenna that increases building complexity by adding a multi-level antenna to rooftop*

passes the complexity check, a random x and z position are chosen on the top of the building, and the antenna is constructed at the building's y-height, as per lines 7-9. Then, since antennae are oft multi-layered and decrease in size as they grow taller, lines 10-15 deal with randomizing the number of layers and decreasing the size per each one. To see how these buildings and anetenna look in practice, Figure 2 shows a grouping of buildings with varying heights and high complexities; all have many windows, rooftop hats, chimneys and antennae.

These buildings worked as my first component for modern American cities. However, not all buildings in modern cities take after the blockiness of those represented in Figure 2; for instance, many American cities have a spread of building ages represented in their makeup, with some buildings dating back from the 19th and 20th centuries. Frederick summarizes one of the defining characteristics of this so-called traditional architecture in tip 60, stating that "Traditional architecture employs a tripartite, or base-middle-top, format... the base is quite heavy and thick; the top... is symbolically a crown or hat that announces on the skyline the building's purpose[6]." To succinctly model this geometry, I created a second set of feature-generating functions, this time aimed at abstracting the tripartite structure. By creating defined bases, individualistic windows, and roofs that embed distinct peaks into the skyline, I aimed to assemble models representing "aged" buildings of the past. Figure 3 shows the results of this abstraction.

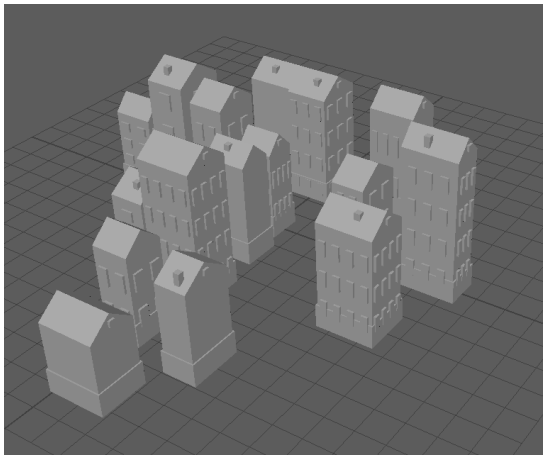

Figure 3: *Abstract representations of traditional buildings*

Having created a procedural generator for both modern and traditional low-poly building abstractions, I had the bulk of what I wanted to be using to assemble my cities. There remained only a few more special structures I wanted to hand-model before proceeding onward.

ii. Special Structures

Throughout history, special structures—like sculptures, temples, churches, and other communal spaces—have served as places of orientation within a city. Such monuments help serve as local landmarks, and cities often evolve around and accommodate for historical monuments. Thus, I wanted to model a few such monuments that could potentially shape how

a city is structured, and inform the orientation and alignments of pathways and buildings.

Since these special structures are few and far between within a city, meaning that they do not require the use of procedural generation to assemble large quantities with slight variations, I used Blender to hand-model them. I modeled three places of congregation: one resembling a temple of Greek or Roman descent, one resembling a modest-sized church, and one resembling a basic tower. I also modeled three sculptures: one monolith, one humanoid, and one quadruped animal.

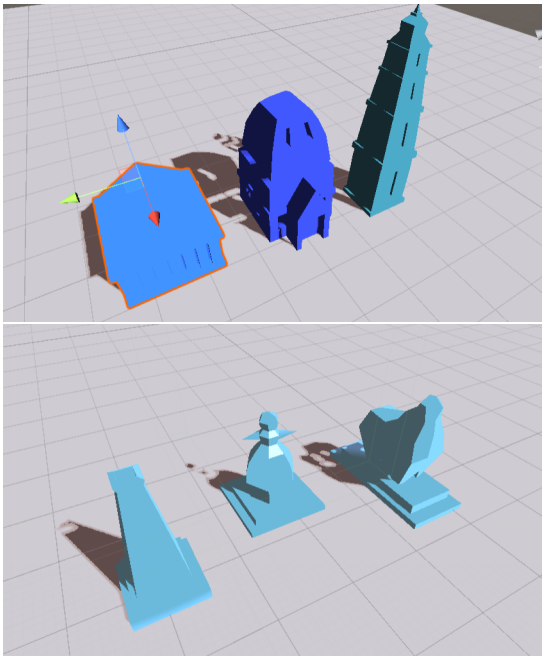

Figure 4: *Top: temple, church, and tower models*
Bottom: monolith, human, quadruped models

iii. City Assemblage

Some of the first computer-aided representations of cities I encountered in my youth came packaged inside of video games. SimCity, Pokemon—though these games promised cities, their cities were unrealistic. Granted, it is fair that the focus of many games that feature cities, like Pokemon, use the term "city" to convey a locale that serves as a hub for quests, items, and characters—and thus, the city is meant to act as a means of story progression, not as any legitimate representation of a real city. But for a game like SimCity, which derives its name from the portmanteau of "simulation" and "city", promising a simulation of a city and failing to deliver upon it is a more egregious oversight. In this section, I will first delve into some of the more notable representations that exist for CG cities currently, and then move to discuss considerations that CG cities aiming for realism may benefit from.

Figure 5: *Pewter City, from Pokemon FireRed and LeafGreen[17]*

We'll begin by taking a look at Pewter City, from the original Kanto region of Pokemon FireRed and LeafGreen[17]. Ignoring the 2D-birds-eye-view affordance of having all buildings face the same direction, there is something else immediately off about Pewter: it is extremely sparse! From a realistic point of view, it is audacious for a town of 6 buildings, each a fair walk away from the next, to call itself a city. However, in the Pokemon sense of the word, Pewter is a bona-fide city; after all, Pewter is home to its own Pokemon Gym, where the player can battle a Gym Leader for their next badge on their quest to capture all the Pokemon in the Kanto region. Pewter City serves its purpose perfectly at that level: it has all the buildings relevant to the player's needs—including the Gym, a Pokemart to resupply at, and a Pokemon Center to heal at—and a means of getting from one to the next. Even

if a city of 6 buildings would be hardly called a town in real life, when contextualized in the game of Pokemon, Pewter is a city through-and-through.

Figure 6: *SimCity, mobile version[18]*

Next, we'll take a look at the more recent iterations of SimCity for mobile devices [18]. Figure 6 shows one such screenshot—and while it may, at a glance, look to resemble the cities we are accustomed to much more closely than Pewter City from Pokemon, it is not without its own visualization errors. For one, these buildings are built atop empty green land—that greenery representing authentic nature before mankind's intrusion. Except, in a land of such dense urban development like the one pictured here, skyscrapers are hardly being built over newly found land. Rather, such development would likely occur in populous areas of already-developed cities: areas deemed worthy of further investment. The surroundings should already be concrete and cement; there shouldn't be grassy land in sight, save for the occasional park. Furthermore, each building appears to be a skyscraper distinct from the buildings surrounding it. It occupies its own plot of land, leaving space between it and the next lot over. However, simply walking through a city will call to attention that most cities are so densely packed, that these sorts of alleyways rarely exist between adjacent buildings lining the side of the road on a given block. To have each building be cordoned off to the point where no two adjacent buildings share a single wall is unheard of.

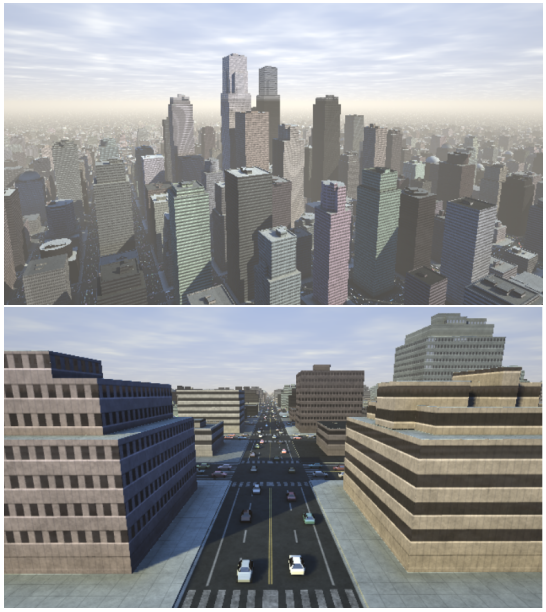

Figure 7: *3D renders of a city via Shadertoy[15]*

SimCity is not the only city generator to have encountered these issues, and its treatment of the issues may be more justified given that there are other gameplay factors driving these issues of representation within the game. Looking at user otaviogood's 3D Skyline render from Shadertoy[15], we see that they run into similar issues: although the first image in Figure 7 makes a compelling case for a generative city from afar, a closer look at the roadways in their representation reveals that blocks are split by roads into equally-sized squares, with each one containing a singular building of varying structure. It looks good from afar and above, but there are fundamental issues with the underlying reasoning behind a choice like this one—namely, it is completely unnatural. No city in the world harbors such a structure, so why does this generative one?

Even if it makes no sense, it's easy to think about, and easy to implement. The first "city-esque" render I made was similar: a large number of buildings situated at coordinate-aligned points on a grid. As seen in Figure 8, from afar, it's dense and city-esque—but that city-like perception is superficial at best.

In reality, cities are made up of streets lined

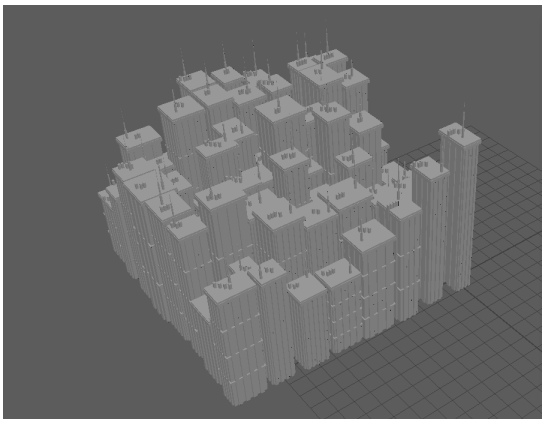

Figure 8: *Render of a basic, gridded city-like*

with buildings that create pathways to navigate an otherwise urban jungle. What I wanted to achieve was urban density, but with the legitimate road and block structure of a real city—unlike Pewter City, which is too sparse and poorly organized, and unlike SimCity and other 3D renders, which look superficially like cities but lack real urban structure. To this end, I decided that the best place to start generating cities would be to generate a road network first. I adopted road-first template-based generation as the method I would be using to generate road structures[10]. Template-based generation entails creating algorithms to generate road structures based off a certain desired configuration, which acts as your "template".

When deciding what my templates would be, I studied a few modern cities—New York City, Boston, and Paris, specifically—to gain an understanding of how their road systems were structured [3, 14, 16]. These three led me to create 3 initial templates for constructing roadways: gridlike templates, as seen all throughout NYC and in other pre-planned cities; radial templates, as seen around the Arc de Triomphe in Paris, and to a lesser degree, other monuments; and random templates, seen interspersed between and across other roadways.

iv. Templates

After exporting the models from Maya and Blender, I moved to Unity to algorithmically construct templates to generate these cities.

Figure 9: *Google Maps view of NYC [14]*

Using NYC as a reference, gridlike templates were implemented by making an algorithm that took parameters for grid width and height, and created two dictionaries: one for vertices, which stored vertex names as keys with their Vector3 locations as values, and one for edges, which stored vertex names as keys with a list of their connected vertices as values. Separate parameters for width and height allow it so there can be variation in the rectangular form of these gridlike blocks, meaning that like NYC, avenues and streets can be at different intervals. Our grid is therefore not confined to squares, making it more robust than coordinate-aligned grid city models. In the assembleGrid method, the loop from lines 1-6 determines and appends each position to the vertex dictionary. Afterwards, adjacent edges are connected in the edge dictionary.

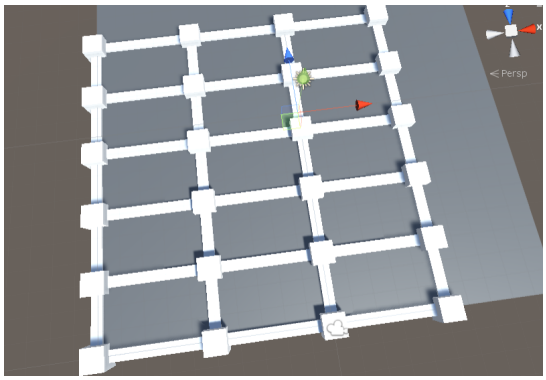

Figure 10: *Grid road structure*

```
1  void assembleGrid(){
2      for (int x = 0; x < gridWidth;
           x++){
3          float xIdx =
               x*(renderDist/gridWidth) -
               (renderDist/2);
4          for (int z = 0; z < gridHeight;
               z++){
5              float zIdx = z *
                   (renderDist /
                   gridHeight) -
                   (renderDist / 2);
6              Vector3 pos = new
                   Vector3(xIdx, 0, zIdx);
7              vertices.Add(""+names[x] +
                   names[z], pos);
8  \\...
9
10 void radialBranches(){
11     for (int x = 0; x < numBranches;
           x++){
12         float angle = (360/numBranches)
               * x + Random.Range(-15, 15);
13         float len = renderDist /numLvls;
14         Vector3 pos = new Vector3(len *
               Mathf.Cos(angle *
               Mathf.Deg2Rad), 0,
               len*Mathf.Sin(angle *
               Mathf.Deg2Rad));
15         vertices.Add(names[0] + names[x]
               + "", pos);
16         radialHelper(numLvls-1,
               numBranches-1, names[0]+
               names[x]+"", angle);
17 \\...
18
19 Dictionary<string, Vector3>
       randomVertices(){
20     for (int x = 0; x < numLocii; x++){
21      \\...
22         Vector3 pos = new
               Vector3(Random.Range
               (-renderDistance,
               renderDistance), 0,
               Random.Range(-renderDistance,
               renderDistance));
23         cube.transform.position = pos;
24         vertices.Add(names[x]+"", pos);
25 \\...
```

Code 3: *Methods to generate city templates*

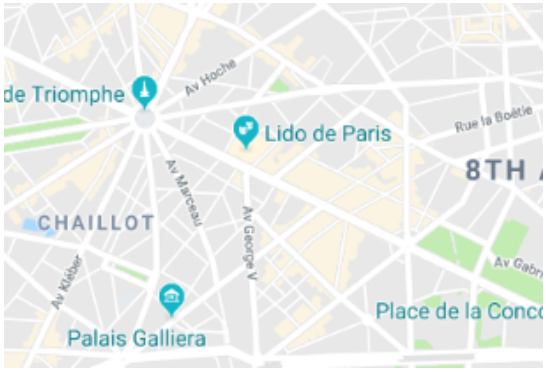

Figure 11: *Google Maps view of Paris [16]*

Based off the roads in Paris surrounding the Arc de Triomphe, radial templates were constructed by first determining how many branches the radial pattern would begin with, and how many levels the radial pattern would continue out for. At each successive branch, the recursive radial helper function called in line x decreases the number of levels and branches until all levels are exhausted, creating a radial tree-like pattern where successive branches shrink in size until dissipating.

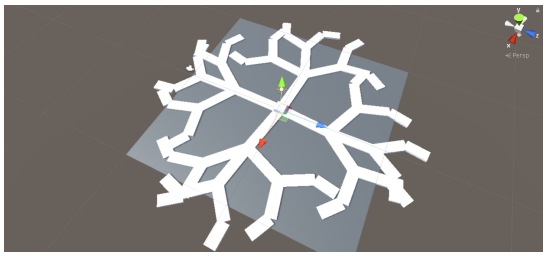

Figure 12: *Radial road structure*

The random templates are not based off any cities in particular. Rather, these templates are intended to have a less regimented structure compared to the grid and radial templates, and include more triangulation, odd angles, and strange partitions. While I am not taking into account natural terrain, as all my city generations have occured on artificially flat planes, random templates help model the strange road structures that sometimes occur from inconvenient hills, waterways, and natural features. The randomVertices method creates a parame-

terized number of vertices at random locations within the given render distance, then creates random connections between said values in the vertex dictionary.

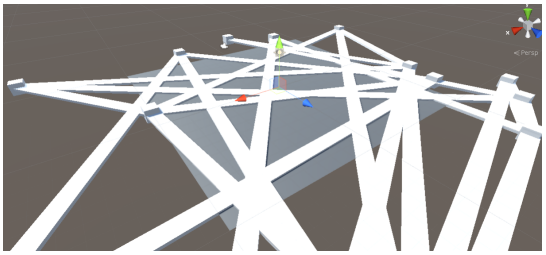

Figure 13: *Random road structure*

Figure 14: *Google Maps view of Boston [3]*

Independently, these templates account for potential portions of cities, but not all cities adhere to a singular template. Looking at Northern Boston, we can see that there seems to be some combination of randomness and gridlike roadways happening[3]. Thus, my next step was to create dual-layer templates that incorporated these templates I had already made, and organized roadways into two divisions: super-structures, and sub-structures. Super-structures refer to the larger roadways that facilitate broad movement across a city, and sub-structures refer to the smaller passages that allow for movement within specific areas of a city. For instance, I would classify this portion of Boston as having randomness in the super-structure, and a gridlike sub-structure. Translated into a dual-layer template, a Boston-like road structure takes on the appearance of Figure 15.

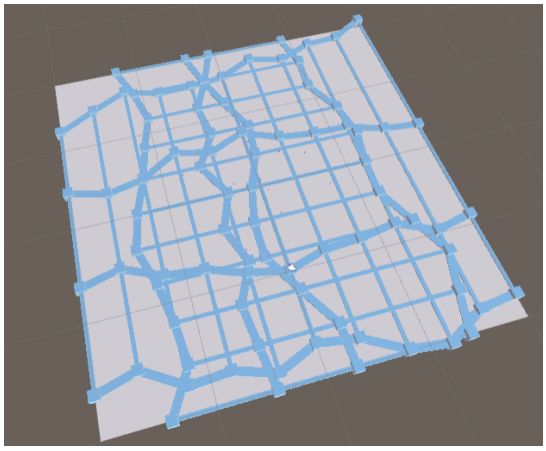

Figure 15: *Boston-like dual-layer road structure*

v. Visualization

Once roadways were constructed and stored in dictionaries, I generated buildings flush against one another along both sides of each road segment, then deleted those that intersected the roads themselves to create some semblance of city blocks. Furthermore, I color-coded buildings to make it more immediately apparent which buildings were modern, and which were traditional. Modern buildings are dyed in sunset shades of orange-red; Traditional buildings are wooden brown, to evoke older materials used in buildings.

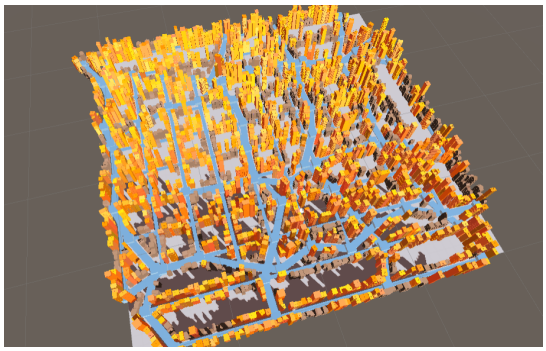

Figure 16: *Boston-like city visualization*

To balance the spread of traditional buildings amongst the modern ones, I created parameters chanceOld and clusterOld to determine the chance of the next building being traditional given that the prior building was either a) modern, meaning that this would be the chance of a traditional building in the midst of modernity, or b) traditional, meaning that this would be a traditional building in a cluster of traditional buildings. Many cities still have both: the occasional historical artifact amongst towering buildings, and the old-town district full of remnants of the past. Figure 16 has some discernible clusters of old buildings, while Figure 17 depicts a quaint, older town with greater separations between each house. As a means

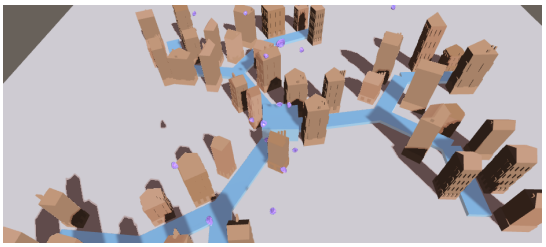

Figure 17: *Radial old town visualisation*

of highlighting the abstract nature of these representations, I coded a toon shader to heighten the non-photorealistic appearance of the visualizations and give them a more cartoony, flatly-shaded feel.

vi. Traversal

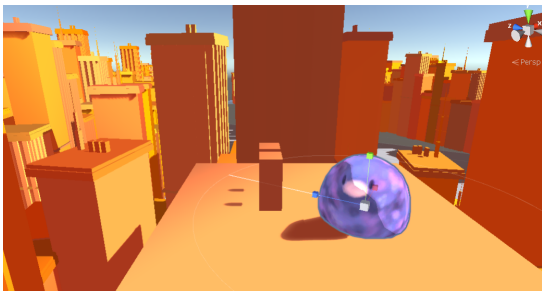

Figure 18: *Pigeon atop a roof*

Up until this point, I had been viewing my cities in the Unity Scene view, for easier orientation and observation. However, I wanted to know what it would look like to traverse the world I had generated from the world's own scale. So, I modeled, textured, and animated a bird, then mounted the player's camera atop said bird. The world became traversable from the back of everyone's favorite city-dweller: the pigeon.

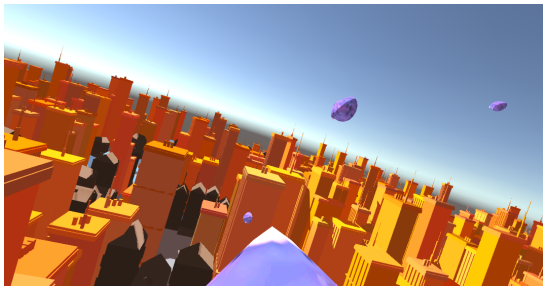

Figure 19: *On the back of a pigeon*

I also created a pigeon-spawner in the middle of any given city that tosses out pigeons spiraling around in random arcs. These generative pigeons lack any sense of collision detection... a behavior real pigeons exhibit, too.

III. Growth

In the second part of my project, I focused on methods of generating 2D skylines. The tool I used to output my own was Shadertoy, a tool for creating and sharing WebGL shaders online. The visualizations done by other community members provided a good starting point for me, allowing me to establish what the canonical skyline representation is, and use that canon as a starting point to determine new ways in which it could be changed to account for social factors relating to growth. Hence, the final goal was to create an abstract representation of the city skyline that comments upon social conditions within cities.

i. Canon

Before we begin to look at reconstructions of skylines via CG shaders, let's first make sure we're on the same page as to what a*real* skyline

Figure 20: *Boston skyline at night [2]*

looks like. Refer to Figure 20, which shows the skyline of Boston at night: brightly lit against the dark, desaturated blue of night[2]. Good? Good.

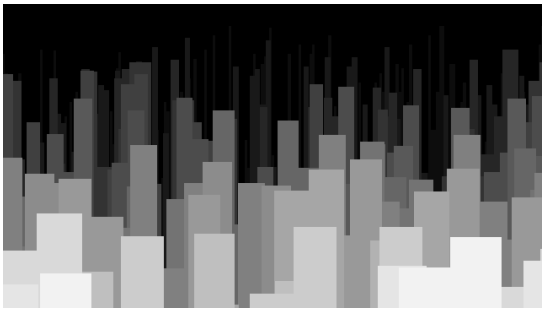

Figure 21: *Canonical visualization of a 2D skyline [9]*

The canonical visualization for a 2D skyline shader is like that seen in Figure 21, made by Shadertoy user gsingh93[9]. Comparing it to a real skyline, we can see that it demonstrates considerations like: multiple layers of buildings with varying colors, getting darker as they recede into the distance; scrolling horizontal movement over time, as if watching the skyline pan by through the window of a train; and varying heights at regular intervals, with each plateau representing a distinct building.

ii. Time

Interestingly enough, when we take a look at the code for Figure 21's representative shader in Code 4, we'll note that the uniform variable iTime is only used once: on Line 6 as a means of making the city scroll across the screen over time. However, cities are subject to numerous changes over time. In fact, the first thing we can do to improve upon this city is acknowledge the passage of time in its

```
1 void mainImage(out vec4 fragColor, in
      vec2 fragCoord) {
2     vec2 p = fragCoord.xy /
          iResolution.xy;
3     float col = 0.;
4     for (int i = 1; i < MAX_DEPTH;
          i++) {
5         float depth = float(i);
6         float step = floor(200. * p.x /
              depth + 50. * depth +
              iTime);
7         if (p.y < noise(vec2(step)) -
              depth * .04) {
8             col = depth / 20.;
9 //...
```

Code 4: *Canonical 2D skyline shader [9]*

```
1 float col = 0.;
2 vec3 color = 0.3 + 0.7 * ((cos
      (iTime/4.+ p.xyx+ vec3(0,0,20))));
3 \\...
4 col = 1.-(depth / 50. + 0.80);
5 color = (color*0.8 + vec3(col)*0.2);
```

Code 5: *Incorporating iTime to create time-sensitive sky and add fog to city*

visualization—-which can be accomplished by turning the background into a gradient of hues that simulate the colors the sky takes during the day, through sunset, into the evening, and so on.

To emphasize the sky and "skyline" nature of the visualization, I adjusted the perceived "angle" at which we are viewing the city, compressing the levels together closer to the bottom of the screen, and changing the colors of the buildings to various shades of grey that get lighter the further away they are—as if being enveloped by urban fog, which makes distant objects harder to see. Code 5 shows how given these updated grey buildings and multi-colored sky, we can fully implement the idea of "fog" by taking a weighted average of the current sky color and the building color for each depth of buildings. Adjusting the

coefficients of the weighting allows us to control the fog's dissipation: weighting the sky color higher means our fog will appear more thick, and further away building depths will be harder to see; and weighting the building color higher means our fog will appear to dissipate, making the faraway buildings stand out clearly against the horizon. The new visualization, seen in Figure 22, has the sky feature more prominently as a reminder of the passing time.

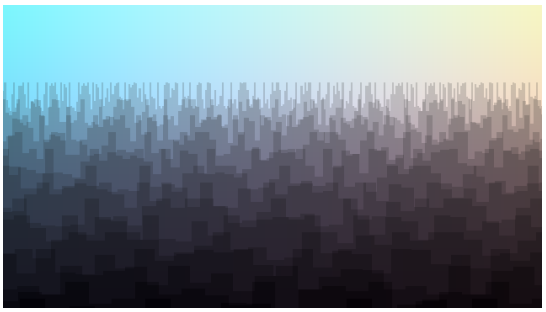

Figure 22: *Visualization of 2D skyline with time-sensitive sky*

iii. Growth

So far, we've begun to account for time at the temporal level. However, we have yet to show what effect that temporality has upon the city's buildings themselves! In fact, when thinking about how cities change over time, the growth of cities cannot be ignored. Urban sprawl and gentrification rule modern-day cities: where urban life can be funneled into and out of areas at the drop of an eviction notice, or an updated zoning code[6, 13].

In order to account for growth over time in the city, we must incorporate iTime when determining the height of a given building, instead of leaving it all up to a random seed. To do so, I wrote a function called height, which turns the shader into a clip of LOOP_INT seconds that demonstrates the growth of a city over time; Code 6 shows the height method, which works by resisting building height randomization at a rate inversely proportional to the runtime of LOOP_INT. The resistance is

```
1 float height(vec2 p, float dep){
2     return 0.1 + .5 * ((1. - dep/
          float(MAX_DEPTH)+iTime/80.) -
          (1. - random(p) /
          ((float(LOOP_INT) -
          mod(iTime,float(LOOP_INT))) /
          3.))) - dep * .02;
3 }
```

Code 6: *Height function for growth in skyline*

greatest at the start, and zero by the end of the (currently) 90-second interval: visualized, the shader initializes the city with all buildings uniformly low—but as time passes, the growth of these buildings begins to spike, as if simulating the gentrification of slums by sudden development of luxury condos. The city gradually increases in the number of buildings it contains by adding more and more layers of buildings, increasing in height, until a sudden explosion of exponential growth envelops at the end of 90 seconds. Figure 23 shows how little growth occurs between the 45- and 70-second screencaps, but massive growth occurs at the 85-second screencap, enveloping the screen[7].

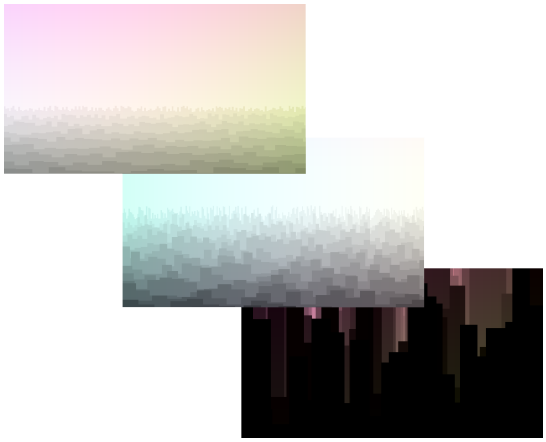

Figure 23: *Screencaps at 45-, 70-, and 85-seconds in 90-second growth clip*

```
float height(vec2 p, float dep){
    return 0.+ .5*((1. - dep/
        float(MAX_DEPTH) + iTime/80.)
        + random(p)/(iTime/5.)) - dep
        * .02;
}
```

Code 7: *Height function for decay in skyline*

iv. Decay

Last, but not least, comes the foil to growth: decay. Just as sudden growth is liable to hit a city and cause buildings to suddenly skyrocket into skyscrapers, urban decay in areas of long-standing neglect are similarly common in large cities.

To simulate this, I adjusted the height function seen in Code 7 such that the randomization now decreases as a function of time increasing, and at a much more slow and steady pace than the growth cycle. As Figure 24 shows, the clip begins with us panning out from right next to a building, leading into a long stretch of buildings slowly starting to flatten out and stop growing, until the city eventually piles up and flatlines beneath stagnant rows of flattened buildings after 180 seconds have passed—though the process continues off-screen, beyond our viewport. After all, decay is an ongoing process[8].

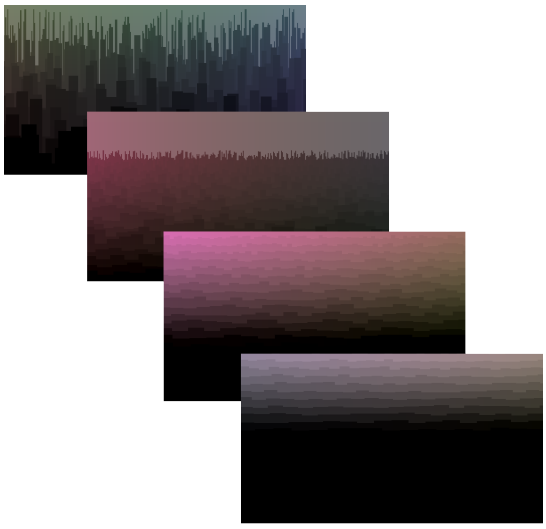

Figure 24: *Screencaps at 5-, 40-, 100-, and 150-seconds in 180-second decay clip*

IV. Conclusion

While procedural generation is by no means a small field, there is still much room for expansion in generative methods that go beyond superficial appearances, and pay more attention to the rationale governing the logic of structures. In my research on methods for generating structure in 3D models and growth in 2D shaders, I encountered several procedural visualizations of cities that lacked the logic to generate what could truly be considered a generative representation of a city.

This room for growth in the area of procedural generation holds not only for environments like cities, but also for more granular generative projects—for instance, like marble textures that simulate the process a metamorphic rock goes through as it crystallizes. However, such a feat is easier said than done. To create accurate representations requires an individual both the means to generate procedurally, and the specific knowledge of whatever is being generated. Likewise, the biggest limitation I encountered while completing my research was that my knowledge of urban design and planning was not as deep as I would have liked; for this project, I focused on researching and implementing methods for procedural generation, applying only basic principles of urban design and planning. However, should I decide to resume working on this project down the line, I would like to devote more time to studying urban politics and design first, and use that to influence and inspire new directions for procedural generation.

In the future, I'm curious to expand upon procedurally generating 3D cities given a specific terrain. Understanding how terrain interacts with the pre-planning for, or gradual development of, a city over time is integral when trying to represent at how real cities have come into being. After all, Rome wasn't generated in

a day.

V. Acknowledgements

I'd like to extend my deepest gratitude to Professor Nick Montfort, who has been my advisor for the duration of this project! His insight has constantly pushed me to think in new directions and try new things, and his encyclopedic knowledge of all things generative-art-related never ceases to amaze me. You're the best! Thanks for dealing with me!!

References

[1] Aracthor. "How to Generate a City Street Network?" *Game Development Stack Exchange*, Stack Exchange Inc, 27 May 2016, gamedev.stackexchange.com/questions/122015/how-to-generate-a-city-street-network.

[2] Baetscher, Eric. "File:Boston Night pano1.Jpg." *Wikimedia Commons*, Wikimedia Foundation, 20 Nov. 2007, commons.wikimedia.org/wiki/File:Boston_night_pano1.jpg.

[3] Boston, Massachusetts, USA. *Google Maps*, 2019, maps.google.com.

[4] Buck, Jamis. "Dungeon Level." *AARG*, AARG Net, 2003,

[5] Buck, Jamis. "Random Dungeon Design: The Secret Workings of Jamis Buck's Dungeon Generator." *AARG*, AARG Net, 20 Feb. 2003, web.archive.org/web/20080203123815/www.aarg.net/m̃inam/dungeon_design.html.

[6] Frederick, Matthew. *101 Things I Learned in Architecture School*. MIT, 2007.

[7] Gerr, Joanna. "Skyline_2D_GrowthOverTime." *Shadertoy*, Beautypi, May 2019, www.shadertoy.com/view/wlX3W2.

[8] Gerr, Joanna. "Skyline_2D_DecayOverTime." *Shadertoy*, Beautypi, May 2019, www.shadertoy.com/view/WlfGD2.

[9] gsingh93. "Skyline Explanation." *Shadertoy*, Beautypi, 23 June 2015, www.shadertoy.com/view/4tXSRM.

[10] Kelly, George, and Hugh McCabe. "A Survey of Procedural Techniques for City Generation." *CityGen: Procedural City Generation*, www.citygen.net/files/ Procedural_City_Generation_Survey.pdf.

[11] Martek, Craig, "Procedural generation of road networks for large virtual environments" (2012). Thesis. Rochester Institute of Technology.

[12] Mikoleit, Anne, and ÌĹPurckhauer, Moritz. *Urban Code: 100 Lessons for Understanding the City.* MIT Press, 2011.

[13] My Brooklyn LLC.; a documentary by Kelly Anderson and Allison Lirish Dean; director, Kelly Anderson ; producers, Allison Lirish Dean and Kelly Anderson.*My Brooklyn. [New York] :New Day Films*, 2012. Print.

[14] New York City, New York, USA. *Google Maps*, 2019, maps.google.com.

[15] otaviogood. "Skyline." *Shadertoy*, Beautypi, 24 Sept. 2015, www.shadertoy.com/view/4tXSRM.

[16] Paris, France. *Google Maps*, 2019, maps.google.com.

[17] "Pewter City." *Pokemon Wiki*, Wikia Inc, pokemon.fandom.com/wiki/Pewter_City.

[18] Whitlatch, Adam. "Opinion: A SimCity View of Education Won't Fix Philly Schools." *PhillyVoice*, WWB Holdings, 4 Apr. 2017, www.phillyvoice.com/opinion-a-simcity-view-of-education-wont-fix-philly-schools/.