Alex DeJong
Armando Roque

L-Systems for Cities

Final Project Report

Instructions for Use

Compilation

This project is using the codebase from Assignments 3 and 4. As such, please use the same commands to compile the code and launch the web server to run the program.

1. Run the make-skinning.py script.
2. Launch the web server.
    a. http-server dist -c-1
3. Open http://127.0.0.1:8080.

City generation controls

The digit keys of the keyboard are used to indicate the size of the city that should be generated. Pressing ‘1’ will generate a 10x10 city, ‘2’ a 20x20, all the way to 90x90 by pressing ‘9’.

Pressing the key digits while also pressing the ‘shift’ key results in changing the type of area being rendered.

- ‘Shift’ + ‘1’: Downtown area.
- ‘Shift’ + ‘2’: Residential area.

Camera controls

The code includes the implementation of camera controls from assignment 3. This means that you can use the following keys:

- W, S: Zoom camera.
- A, D: Translate (if in fps mode) or swivel (if in orbit mode) camera horizontally.
- Up and down arrows: Translate (if in fps mode) or swivel (if in orbit mode) camera vertically.
- C: Switch between fps and orbit mode. Orbit is initially active.
- Left click mouse drag: Orbit target (if in orbit mode) or translate target (if in fps mode).
- Right click mouse drag: Zoom camera.

Demos

Presentation video of unfinished project: here.
Demo video of final product: here.

Introduction/Abstract

The purpose of this project is to design an algorithm that is capable of procedurally generating different city landscapes based on a number of settings provided by the user. The main goal of using this system is to allow for dynamic and diverse on-demand landscape generation. Because of procedural generation, every generated landscape is different.

There are two main components to this city generation system: a road generation system, and a building generation system. The road generation system is an L-system that creates streets based on a set of action rules and their probabilities (the user provides the probabilities for each rule). The building generation system creates cuboids with random heights and random textures to provide a sense of architectural diversity in the landscape. The style of the generated models for both systems depends additionally on the area type parameter that the user provides. Currently there are two supported area types: downtown and residential.

WebGL is used to render the city landscape in a web browser after the city model has been created. More specifically, the codebases from Assignments 3 and 4 are leveraged in order to, among other purposes, allow for camera controls through the mouse and keyboard.

Background

The following L-system concepts were taken from the lecture “L-Systems and Particle Systems” by Dr. Sarah Abraham from The University of Texas at Austin for CS354.

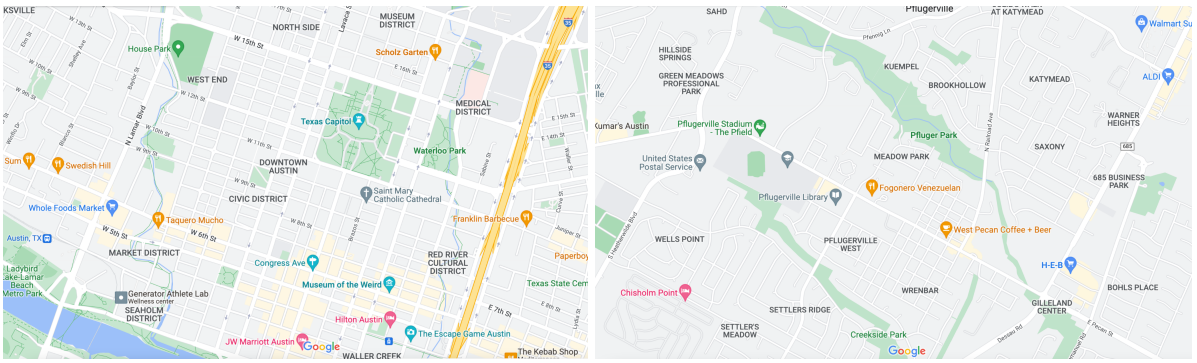
A Lindenmayer system (L-system for short) is a recursive definition of an object using a string rewriting system and formal grammar. It was initially designed to model plants, but was later introduced to computer graphics.

An L-system definition consists of four major elements:

1. An axiom: a starting string.
2. Variables: a set of symbols to be rewritten according to rules.
3. Terminals: a set of symbols that have no rewriting rules.
4. Rules: a set of substitutions possible for variables.

The way that L-systems are implemented in graphics systems is by associating actions that have a graphical meaning (drawing a line, rotating the cursor, etc.) to each variable and terminal. The axiom is expanded n times, and then the actions from the resulting string are executed, which can be interpreted as an image.

L-systems have been extended to a number of use cases, one being the generation of urban landscapes. Sprawling cities have great diversity in their street layouts, but also follow certain guidelines based on the purpose of the grounds (a downtown commercial area is different from a suburban residential zone). The following images illustrate how two areas with different purposes in the same city have different layouts.



Images 1.1 and 1.2.
Image 1.1 is Downtown Austin, note how the grid-like layout is present throughout the whole area.
Image 1.2 is Pflugerville, a suburb of Austin. Note the presence of bigger green areas and that there is no big grid connecting all the streets.

We wanted to create our own city generation engine, our main motivation being curiosity for discovering how city layout guidelines could be encoded into an L-system. We believe that such a system can be very useful for simulation purposes, as well as for video game map creation.

We mainly based our work on two academic papers: “Procedural modeling of cities” by Yoav I. H. Parish and Pascal Müller, and “FL-system : A Functional L-system for procedural geometric modeling” by Jean-Eudes Marvie, Julien Perret and Kadi Bouatouch.

Technology stack

Because we wanted to render the city in 3D and give the user the ability to fly through it, we decided to use WebGL. This decision allowed us to leverage the existing codebases from Assignments 3 and 4. This meant that our codebase is entirely JavaScript, TypeScript and GLSL.

Road generation system

When designing the algorithm for the generation of the roads layout we considered two main approaches. The first one consisted of having an initial block split it recursively, alternating between horizontal and vertical splits. This would have been a very simple mechanism but we thought it would have limited the generated streets to be very “squared” or grid like.

The approach we decided to follow consisted in having what we called “road builders”. These are like snake heads that at any given time have a position on the map and choose an action that takes them to their next position, which is an adjacent location. These road builders have a sense of orientation, meaning that if they take the action “continue straight”, where they move towards depends on their orientation. As a road builder moves through the map, the path it traverses becomes a street. Snippet 1 shows the attributes of the class created to represent road builders.

```
class RoadBuilder {
 public direction: Direction;
 public x: number;
 public y: number;

 ...
}
```

Snippet 1. The RoadBuilder class.

The road creation algorithm

The main idea behind this road generation algorithm is simple: expand the road iteratively using a stochastically chosen rule for the next step at each iteration, given a set of constraints that the road must have in order to be valid. The constraints are there to try to make the layout as similar as how real layouts are.

There is an initial road builder placed at the south end of the city, centered horizontally, looking towards the north. This is the starting road. The initial road builder is put in a queue. Then, in a loop that runs while the queue is not empty, the road builder at the front is popped (referred to as the main builder in each iteration) and that main builder's next action is chosen based on the available action rules and their probabilities. If the new position resulting from the main builder taking the chosen action is valid, that position is marked as a street in the city map (which consists of city nodes, represented by the class shown in Snippet 2), and the builder's current orientation is recorded as well (the orientation of the street is important to render the orientation of the road textures correctly). Additionally, if the new position is valid, the main road builder is then put back on the queue. Putting it back on the queue means that the road will continue expanding in that road builder's path; else if it were not put back, the road that that road builder had constructed so far would stop at its last position.

```
class CityNode {
 public type: GroundType;
 public direction: Direction

 ...
}
```

Snippet 2. The CityNode class.

Then, it is randomly determined whether or not other road builders should be created off (branch out) of the main road builder's previous position (the one it had when it was popped from the queue). This is analogous to the snake growing more heads that will build other roads themselves. If there are any new road builders created, they are put in the queue.

In order to allow each road builder to move "in parallel", they are expanded through the queue. We initially had a depth-first search approach (which would be a recursive call), but it

resulted in having one main road that was completely built before the secondary road builders that branched out at many different points could even start expanding. We thought using a breadth-first search would allow the streets to grow in a more complex and seemingly natural way.

The whole process just described is encoded in Snippet 3.

```
// Method that procedurally builds roads according to rules.
createRoads() {
  // Starting road
  this.queue.push(
    new RoadBuilder(Math.floor(this.size / 2), 0, Direction.North)
  );
  this.map[Math.floor(this.size / 2)][0].type = GroundType.Street;
  this.map[Math.floor(this.size / 2)][0].direction = Direction.North;

  // Build loop.
  while (this.queue.length > 0) {
    const mainBuilder: RoadBuilder = this.queue.shift()!;

    // Save current builder state.
    const prevMainBuilderState = new RoadBuilder(
      mainBuilder.x,
      mainBuilder.y,
      mainBuilder.direction
    );

    // Stochastically choose rule for next action.
    const nextAction = this.rules.getMovementAction();
    this.applyBuilderAction(mainBuilder, nextAction);

    // If the main builder has stopped and its state has not changed, then there
is no need to check for position validation.
    if (!mainBuilder.equals(prevMainBuilderState)) {
      if (this.isValidRoadPosition(mainBuilder)) {
        // It is a valid road position.
        this.queue.push(mainBuilder);
        this.map[mainBuilder.x][mainBuilder.y].type = GroundType.Street;
        this.map[mainBuilder.x][mainBuilder.y].direction = mainBuilder.direction;
      } else {
        // Check if there is a no dead end rule.
        if (!this.rules.allowDeadEnd()) {
          // Try to apply the other rules.
          const availableActions = this.rules.getAvailableMovementActions();
          for (const action of availableActions) {
            mainBuilder.copyState(prevMainBuilderState);
```

```
            this.applyBuilderAction(mainBuilder, action);
            if (this.isValidRoadPosition(mainBuilder)) {
              this.queue.push(mainBuilder);
              this.map[mainBuilder.x][mainBuilder.y].type = GroundType.Street;
              this.map[mainBuilder.x][mainBuilder.y].direction =
mainBuilder.direction;
              break;
            }
          }
        }
      }

      // Branch out to sides.
      // First try to branch out to the left.
      if (this.rules.shouldBranchOut()) {
        this.branchOutToSide(prevMainBuilderState, Side.Left);
      }
      // Now try to branch out to the right.
      if (this.rules.shouldBranchOut()) {
        this.branchOutToSide(prevMainBuilderState, Side.Right);
      }
    }
  }
```

Snippet 3. The main body of the road generation algorithm.

Rules specification

The system currently uses a fixed set of rules for specifying how the road builders should behave. However, since the rules are selected randomly during runtime, it is the probability of each rule that determines the nature of the generated city landscape. The user has full control over what probability each rule has. The user can easily change the probability of the rules in a JSON file. Right now there are two types of layouts available: a densely populated downtown area, and a more suburban residential area. The rules and their probabilities are shown in Snippet 4.

```
[
 {
   "type": "Downtown",
   "movementActionRules": [
     {
       "action": "CS", // Continue straight.
       "probability": 0.8
     },
     {
```

```
      "action": "TR", // Turn right.
      "probability": 0.1
    },
    {
      "action": "TL", // Turn left.
      "probability": 0.1
    },
    {
      "action": "STOP",
      "probability": 0.0
    }
  ],
  "branchOutRules": [
    {
      "action": "doBranchOut",
      "probability": 0.2
    },
    {
      "action": "doNotBranchOut",
      "probability": 0.8
    }
  ],
  "crossIntersectionRules": [
    {
      "action": "doCrossIntersection",
      "probability": 0.5
    },
    {
      "action": "doNotCrossIntersection",
      "probability": 0.5
    }
  ],
  "deadEndRules": [
    {
      "action": "doAllowDeadEnd",
      "probability": 0.0
    },
    {
      "action": "doNotAllowDeadEnd",
      "probability": 1.0
    }
  ]
},
{
  "type": "Residential",
```

```
  "movementActionRules": [
    {
      "action": "CS", // Continue straight.
      "probability": 0.95
    },
    {
      "action": "TR", // Turn right.
      "probability": 0.01
    },
    {
      "action": "TL", // Turn left.
      "probability": 0.01
    },
    {
      "action": "STOP",
      "probability": 0.03
    }
  ],
  "branchOutRules": [
    {
      "action": "doBranchOut",
      "probability": 0.15
    },
    {
      "action": "doNotBranchOut",
      "probability": 0.85
    }
  ],
  "crossIntersectionRules": [
    {
      "action": "doCrossIntersection",
      "probability": 0.05
    },
    {
      "action": "doNotCrossIntersection",
      "probability": 0.95
    }
  ],
  "deadEndRules": [
    {
      "action": "doAllowDeadEnd",
      "probability": 1.0
    },
    {
      "action": "doNotAllowDeadEnd",
```

```
      "probability": 0.0
    }
  ]
 }
];
```

Snippet 4. The rules and their probabilities for the city landscapes.

To come up with these probabilities we looked at existing downtown and residential areas with Google Maps, and through reasoning (and much experimentation) we figured out that these numbers were the ones that resulted in a considerable proportion of generated landscapes that looked similar to what we were expecting (i.e. similar to the real maps).

During experimentation we used a simple method that printed in the console the generated layout. Images 2.1 and 2.2 show how a downtown and a residential layout look in the console, respectively.

Images 2.1 and 2.2.
Image 2.1 represents a downtown layout.
Image 2.2 represents a residential layout.
In both images the asterisks (*) represent the streets in the map.

Rule selection

Whenever a rule is required in order to make a decision about the next state of the layout, the function ‘getActionFromRules’ is called. This function takes a set of rules (which are the candidates for being used in the expansion) and picks one based on a number generated by Math.random(). Our only concern regarding this mechanism is to what degree is Math.random() generating “truly/mostly” random numbers. This is encoded in Snippet 5.

```
getActionFromRules(rules: RoadRule[]): string {
  const prob = Math.random();
```

```
  let accumulatedProb = 0.0;
  for (let rule of rules) {
    accumulatedProb += rule.probability;
    if (prob <= accumulatedProb) {
      return rule.action;
    }
  }
  throw new Error("Rules do not add up to at least 1.");
}
```

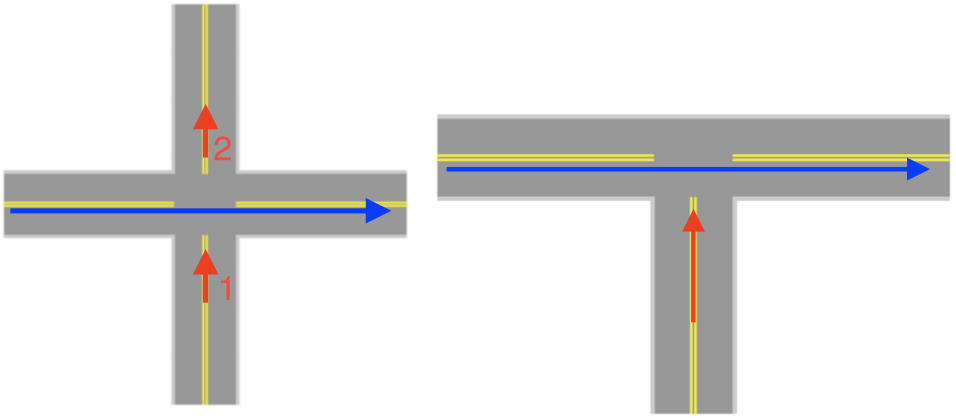
Snippet 5. The function that takes a list of rules and picks one based on a random number.

Constraints enforcement

Whenever any road builder takes an action which results in it moving to another position in order to expand the road to that location, a series of conditions are checked to determine whether a road can be placed there.

The system currently enforces a number of conditions, which determine that a new road's position is valid unless:

- It is outside of the map boundaries.
- The new position is already a street. This means the road builder has reached an intersection. If the rules determine that the road builder should cross the intersection, the system checks if this is possible and if so, places the road builder across the intersecting road, effectively forming a cross intersection (see Image 3.1). If it is determined that the road builder should not or cannot cross the intersection, then a T intersection is formed and the road builder is deleted (see Image 3.2).
- There are existing streets bounding the new street position. This means that relative to the orientation of the road builder, there should not be streets present to the right or left of the new street. There are a few exceptions to this constraint, which account for special types of intersections.
- There is a rule saying that there should not be dead ends. this means that the road builder should try its other action options instead of just stopping and leaving a dead end road.



Images 3.1 and 3.2.

Image 3.1 represents what happens when a road builder (red, first position marked with ‘1’) comes across an existing street (blue) and it is determined that the red road builder should cross the intersection, effectively forming a cross intersection. The red road builder is placed across the intersection, resulting in the position marked with ‘2’.
Image 3.2 represents what happens when a road builder (red) comes across an existing street (blue) and it is determined that the red road builder should not or cannot cross the intersection, effectively forming a T intersection.

How is this an L-System?

While we are not explicitly writing a string as a result of the L-system, the abstract mechanics of the algorithm are an L-system because of the way the rules are chosen when deciding how to expand the existing roads.

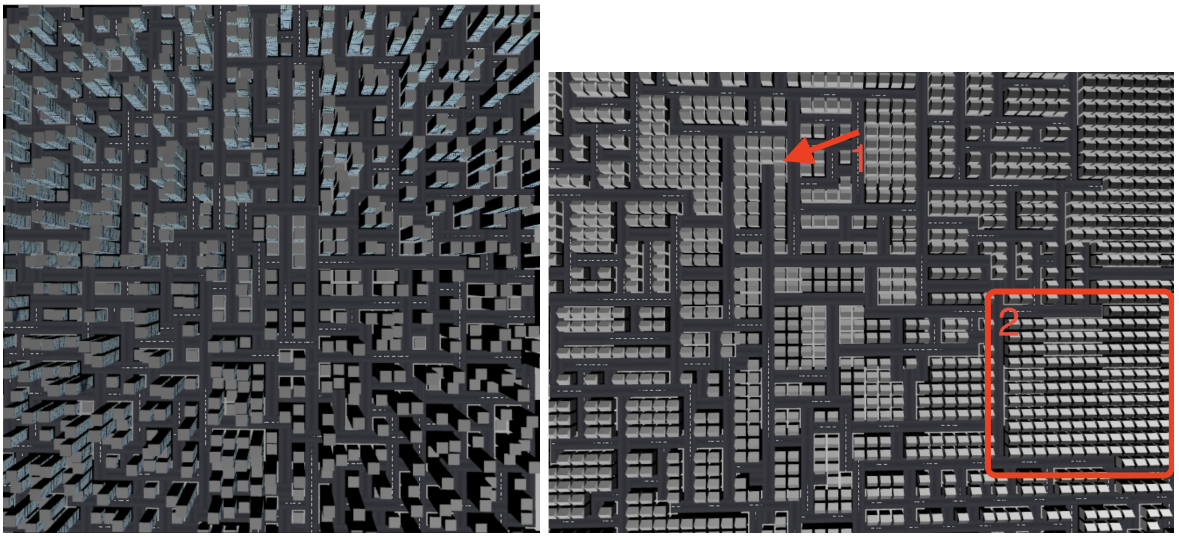
The following is a description of how each element of the algorithm fits within the definition of an L-system.

1. An axiom: the initial road builder placed at the south of the city facing north.
2. Variables: the intermediate positions that a road builder goes through before reaching its final one.
3. Terminals: the last position the road builder reaches before being deleted.
4. Rules: the rules that determine what actions the road builder can take at each iteration of the algorithm’s loop.

Additionally, we expand the L-system with the enforcement of constraints, an idea we got from the “Procedural modeling of cities” paper by Parish and Müller.

Results

Images 4.1 and 4.2 show the results of generating a downtown and a residential area, respectively, and rendering them in WebGL.



Images 4.1 and 4.2.

Image 4.1 is a rendered city landscape using the “Downtown” rules. Note how there are small blocks.
Image 4.2 is a rendered city landscape using the “Residential” rules. Note how there are overall less streets, dead ends (marked by ‘1’), and bigger blocks which could be interpreted as big parks (marked by ‘2’).

Building Generation and Features

The buildings were originally going to be designed using L-systems, but looking at previous research, the complexity of the L-systems to generate 3D geometrical buildings seemed out of scope of what we intended to focus on. Instead, we took the approach of generating a variety of buildings that had different attributes to make a more realistic scene. Based on city type, we created residential buildings or downtown buildings (houses or skyscrapers).

WIth so many buildings, the geometry did not have to be complex to create a cohesive scene. The skyscrapers were simple rectangular prisms, and the houses were just cubes with a small roof. Only a few changes were needed to make a pretty cityscape. First off, the height of the skyscrapers were varied. Textures such as windows were mapped to the building walls to give some color. Similar to the previous assignments, the vertices, normals, and index values were passed to the RenderPass using triangles to draw the geometry.

A light was added to revolve around the city and provide interesting angles to view the buildings from, as can be seen in Images 5 and 6.

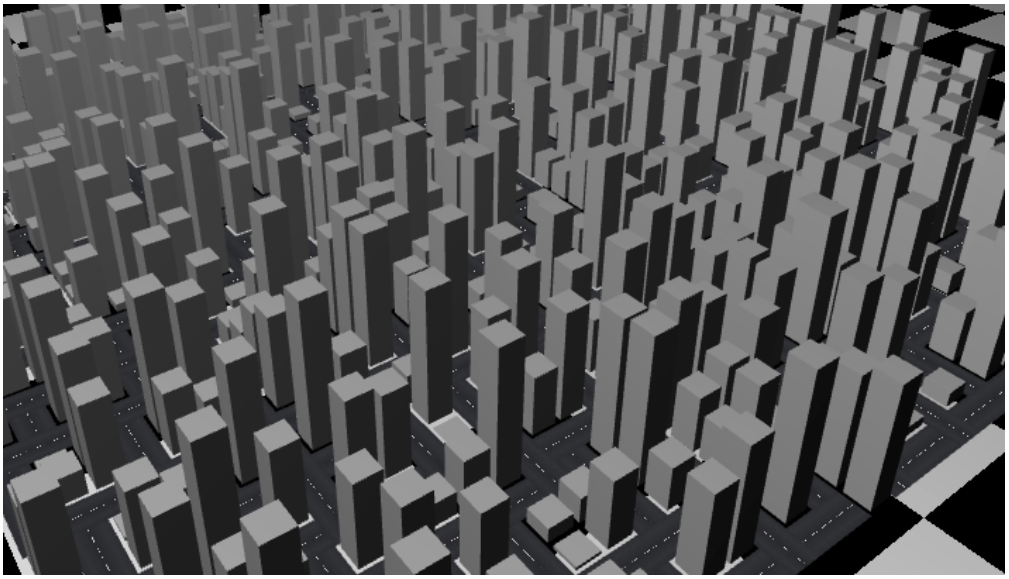


Image 5. A “Downtown” area showing the shadow on the sides of the buildings that face the camera.

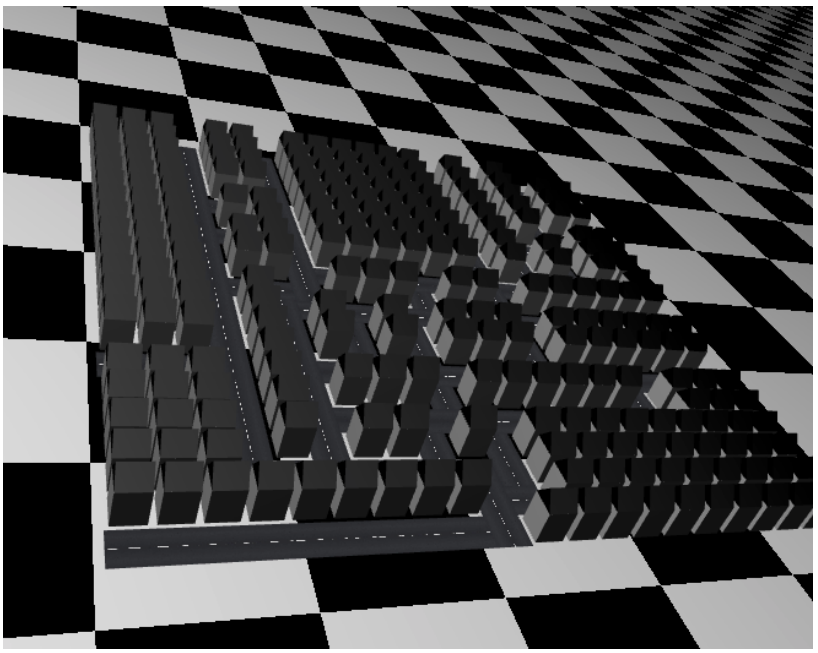


Image 5. A “Residential” area showing the shadow on some of the houses.

We have two methods of rendering, one with a global renderpass for all buildings, and then an individual renderpass per building. The individual renderpass gives more control over building features like texture, but it becomes significantly slower to render.

We also have a “street view” mode by having an FPS toggle. It makes the scene much more interesting when you can explore the city by foot.

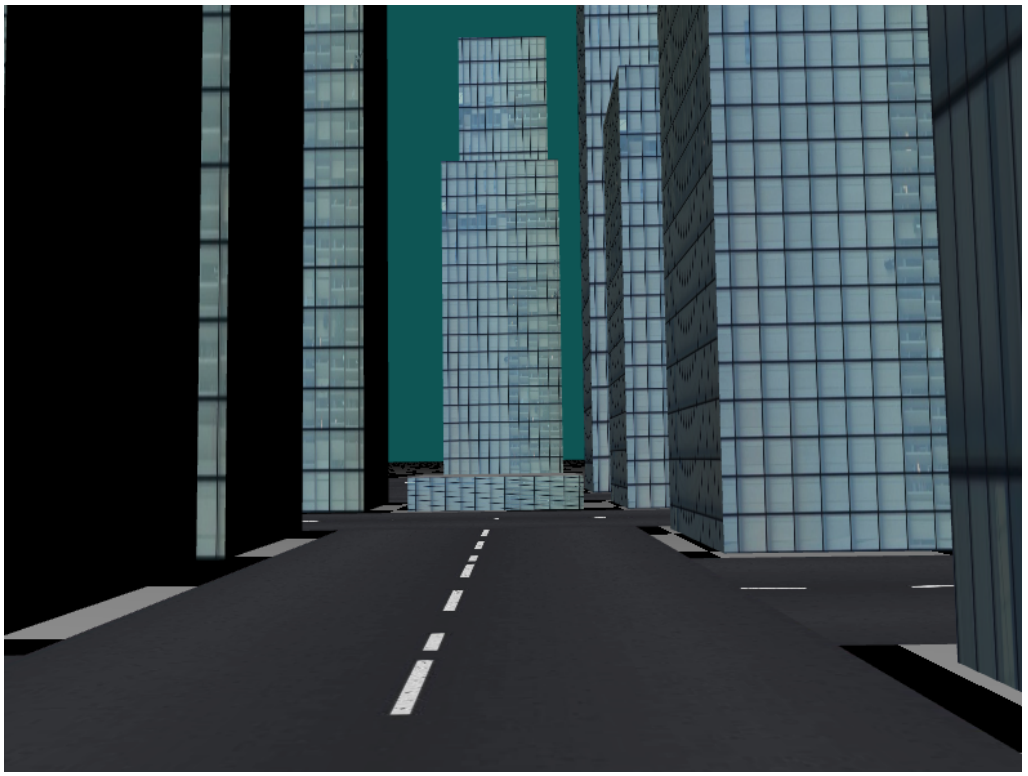


Image 6. A “Downtown” area explored with the fps modality of the camera.

Keyboard controls were also added to quickly switch between city sizes and city types using the number keys. In an instant, the city can expand.

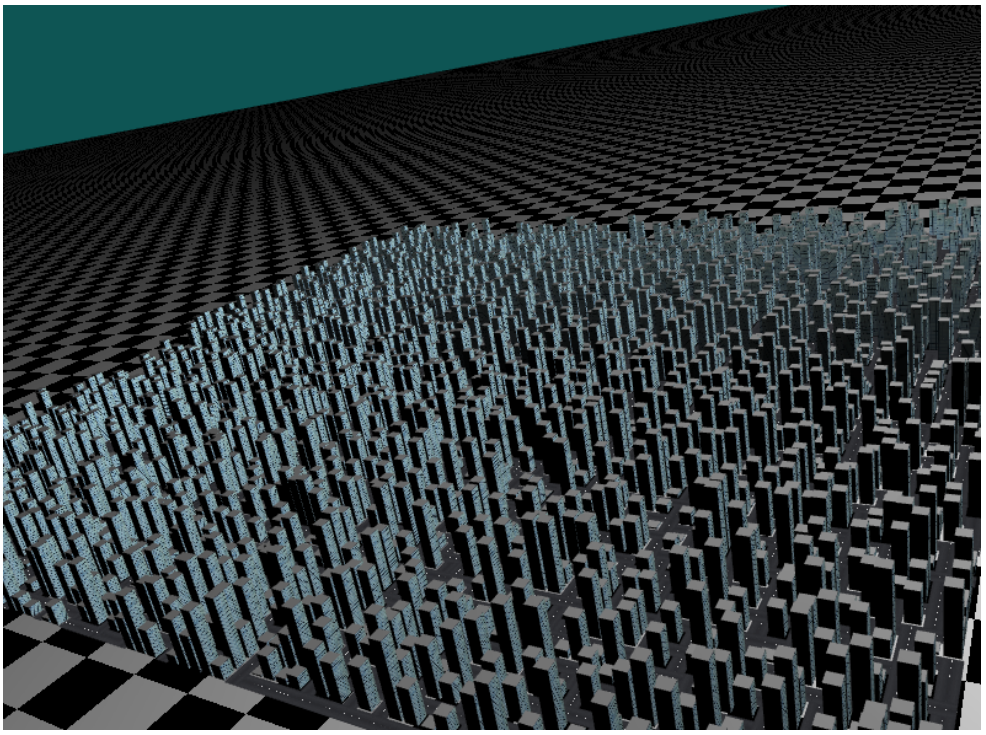


Image 7. A “Downtown” area of great size.

Future Work

Given the scope of the problem and diversity of real-life cities, there is a lot more work that can be done to make more realistic scenes. If we had more time, we would like to build a library of different building, land, and neighborhood types. Parks, bodies of water, parking lots, etc. could all be added to provide more interesting maps and land features. We would also like to incorporate different neighborhood types into a single scene as well. Having different building structures in the same map would provide some contrast between the regions.

We also had plans for randomly grouping buildings in order to make them have different textures within the same scene, meaning that there would be skyscrapers with different styles. We did not have enough time for this, but we think it would be a nice addition to provide more diversity in the landscape.

Adding building shadows would be another interesting feature. Given the density of cities and the large buildings, shadow effects would be prominent in the scene.

Limitations

Road generation

Because of the relative simplicity of our road generating algorithm, we currently have only horizontal and vertical roads. Real life streets go in all directions and do not only turn 90 degrees.

Although the constraints are good to create somewhat reasonable layouts, they are not enough to prevent some roads from being placed in positions where they are senseless. Also, it is somewhat hard to come up with truly suburban layouts without enforcing more complex constraints regarding the size of blocks, spacing between streets, etc.

Finally, because of the nature of random numbers generated by a computer, there are cases in which the created layout is useless, mainly because early in the algorithm a condition determined that one of the main road builders was not in a valid position, effectively terminating the road generation stage.

Building generation

We ran into some problems that limited our project. The RenderPass only saves one texture at a time, making it hard to have multiple textures across the city. One implemented solution was to provide each building with a unique RenderPass, but that made loading and moving around the city with the camera slow and unwieldy. However, a city with just one type of building looks a little bland. One solution to this would be grouping buildings by texture, and providing a renderpass for each individual texture to find a middle ground between diverse buildings and speed.

Another limitation was the difficulty in creating geometries. If there was a way to create more interesting and complex buildings easily without having to deal with vertices, normals, etc.. it would allow for the generation of different houses/skyscrapers.

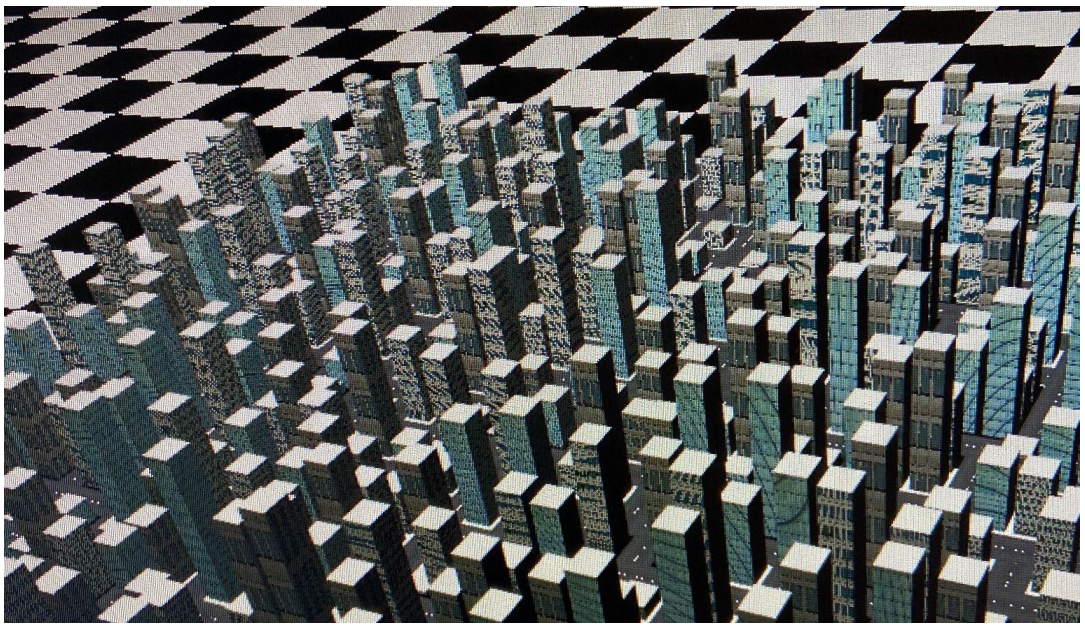


Image 8. A “Downtown” area with buildings using different textures. The issue with this approach was that it was too slow as each one had its own render pass.

References

Abraham, S. (2022). *L-Systems and Particle Systems* [In-class lecture]. CS354 at The University of Texas at Austin, Austin, Texas, United States.

Marvie, J.-E., Perret, J., & Bouatouch, K. (2005). *The FL-system: a functional L-system for procedural geometric modeling*. In The Visual Computer (Vol. 21, Issue 5, pp. 329–339). Springer Science and Business Media LLC. https://doi.org/10.1007/s00371-005-0289-z

Parish, Y. I. H., & Müller, P. (2001). *Procedural modeling of cities*. In Proceedings of the 28th annual conference on Computer graphics and interactive techniques - SIGGRAPH ’01. the 28th annual conference. ACM Press. https://doi.org/10.1145/383259.383292