Western Washington University – CS Department
CSCI 493 Senior Series
Final Report – Winter 2018

Procedural Content Generation

Preston Carroll, Emmanuel Harley, Scott Spitzer

Abstract

Artificial Intelligence requires a lot of data in order to develop an accurate model. For human workflows, this is difficult as it is expensive to generate tags for existing videos, and collect a dataset of the same action done in different environments. The best way to do solve this problem is to procedurally generate the environment with a set of rules that define the necessities of the scene. This project did just that. In order to provide a varying environment for simulations to take place, our group used Prolog to generate layouts as answer sets. From this, we were able to develop kitchen layouts that were both feasible and distinct.

1. Introduction

1.1. Motivation

Currently, datasets for machine learning models to observe human actions are quite minimal. There is not a lot of content for these models to learn from. Part of the problem is tagging videos and processing them involve other machine learning models. Image segmentation, for example, is costly in terms of time and money. The alternative to this is generating the data synthetically, to automate this tagging process with much greater accuracy. This project pursues this goal; to create a platform that provides synthetic data capable of demonstrating multiple examples of human workflows while maintaining compatibility with AI models by supplying the proper tags and filters to videos.

This project handled a specific part of that goal, namely procedurally generating environments in order to create variety in each workflow example. It has some overlap with the responsibilities of tagging data, as the environment must be aware of the content.

1.2. Problem

Environments need to be procedurally generated providing the necessary components to execute the workflow while maintaining a degree of validity and believability.

To properly train AI in human workflow, the environment in which the workflows take place need to be variable. The environment needs to be variable enough to that the AI doesn't just associate the workflow with the environment in a 1-1 correlation, but similar enough to give an idea of the type of environments where the workflow will take place. Environments need to be procedurally generated providing the necessary components to execute the workflow while maintaining a degree of validity and believability.

1.3. Challenges

The scenes need to contain all the necessary components and be flexible enough to apply to multiple workflows and environment types while maintaining believability. Believability is a difficult concept to test and putting an emphasis on it is crucial to the project. Artificial Intelligence is designed to recognize patterns and our procedural generation needs to prevent the specific scene from influencing the pattern recognition of the workflow. Varying scenes randomly would be easier, but doesn't have any guarantees on the rules surrounding the construction of the scene. This is where a major difficulty lies. Generating a scene with necessary and unnecessary components in a manor which is both feasible and variating poses many problems. There are a variety of different objects that interact with each other in numerous ways. Our system must be able to handle this assortment on a broad scale while still taking into account the specific attributes of the individual items.

1.4. Objective

Using both prolog for the algorithm and unity for the scene population, the project will construct diverse scenes in which workflows can take place. The placement algorithm will follow a relation-based model to determine the locations of objects. Objects will be placed in relation to other objects. For example, the fridge can be placed in front of the right wall. To the left of the fridge is a counter. To the right of the fridge is the sink. With this relational model in place, Unity is able to convert a series of relation-object pairs into coordinates within the scene. The resulting scene must not mirror that of previous runs and allow the designated workflow to take place.

1.5. Potential Impact

There is always a need for more data in the world of AI. AI must be trained and tested, therefore requiring a large amount of data. The problem is that data collection can be time consuming and expensive. Quality data even more so. For this reason, research towards generating these examples synthetically is being pursued. Having this synthetic data available will provide AI engineers the material they need to help build models about humans, the environment, and how these things relate with one another. More specifically, the procedural generator developed in the project be be beneficial to the research project underway at WWU focused on training AI to understand human workflow.

2. Related Work

2.1 Key Concepts

Procedural Content Generation (PCG): A technique in computing to create content given a set of predefined rules that will allow for several different permutations of the same content to be made. There are several techniques developed to accomplish this. Some of the more popular ones include L-systems and cellular automata.

Ontology: Hierarchy of objects defined via their properties and their relationships to other objects. We use an ontology to generalize objects into a hierarchy of categories. This allows for broad definition of rules that apply to numerous objects that share a common quality. These qualities are then used to define the kinds of relationships possible between them. For instance, Items are capable of being put inside of Containers. Objects with these properties are allowed to have a relationship of "inside of" and "contains". An illustration of this can be found below.

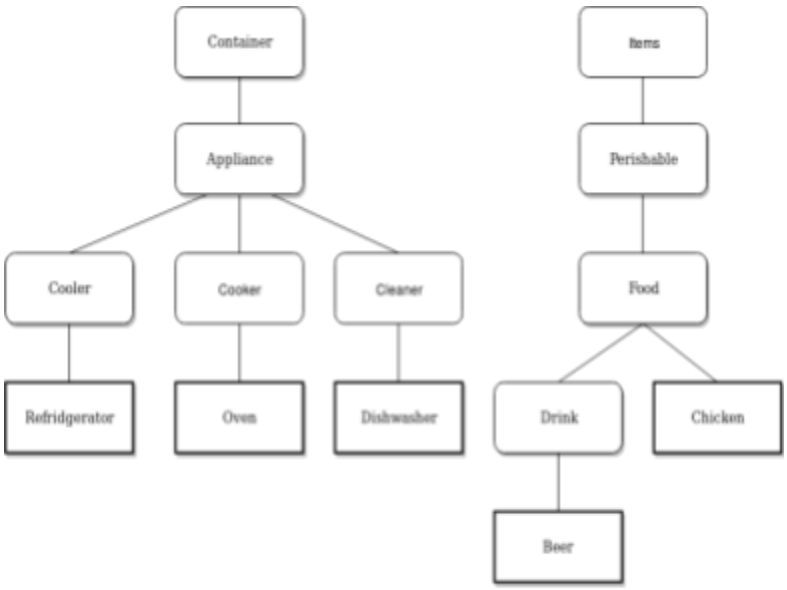


Fig 1: Ontology definition for kitchen objects.

2.2 Related Work

There was a team who explored PCG using prolog as well. They succeeded in using clingo, a version of prolog, to procedurally generate dungeon layouts given a set of dungeon rules and a maze-like algorithm. They developed a set of rules which define a valid layout and a range of values on which to test the rules. Using these tools, prolog was able to generate sets of valid boards [1]. In terms of using prolog to generate layouts in three dimensions, it appears this project is the first.

There are several new groups working on providing a simulated environment to facilitate human workflows [2] [3] [4] [5] [6] [7] [8]. Many of them are using simulations to provide a testing platform for automated vehicles [2] [3] [4] [5] while others are trying to create a platform to facilitate actions, like Virtual Home [8].

The research done by Virtual Home is similar to this project. The group behind Virtual Home has demonstrated the ability to use multiple premade environments in order to facilitate a sequence of human actions. Although they are not procedurally generating the environment, they are using it for the same purpose.

2.3 Key Technologies

To help facilitate the creation of a procedural environment, we sought after a declarative programming language. After a lot of searching and testing, we decided to proceed with Prolog. Prolog allows us to define a set of relational rules and an ontology then query them for a valid state. Prolog is a common tool to manipulate and query ontologies. So it was a natural fit for the job. There were many potential Prolog interpreters we could have used for this project. Unfortunately, one we liked, Clingo, did not follow all the conventions of Prolog and was not documented well enough.

Instead we narrowed it down to two other interpreters: SWI and C# Prolog. C# Prolog was beneficial because it ran entirely in C# and would be much easier to integrate with Unity. The problem was that the interpreter was incomplete. It lacked many standard functions we required and was completely undocumented. At times, we had to edit the source code for the interpreter and add the functions ourselves. This caused some problems because the functions were added didn't have the rollback capabilities inherent to Prolog. Initially, we used C# Prolog was chosen because we couldn't find a method to integrate SWI with an external tool like Unity. SWI runs via an interactive console which adds an unnecessary and undesired step to the process. But, we found a way to compile a prolog program into an executable. Once we determined how to do this in an effective manner, we switched back using SWI.

SWI-Prolog is preferred as it has a lot of support by the community, great documentation, and can compile prolog programs for external tool use. The compiling feature made it simple to create programs relevant to certain tasks in the prolog procedure.

3. Architecture

2.1. Requirements or Use Cases

- Generate initial states:
  - Without manual input
  - Satisfies necessary constraints
  - Believable
  - In a time efficient manner

    - Extensible and integrated into a human workflow environment
- Initial states will:
    - Provide access to all objects
    - Have connected rooms
    - Have no colliding objects
    - Be unique and random once they are finished generating
    - Contain smart objects that agents can interact with
    - Contain all the objects that are given to the generator

2.2. Architecture or Design Space

There are two main components to the design: Unity and Prolog. Unity has the initial and final roles, while prolog does much of the middle work. Unity first has a collection of required objects to populate the scene and generates a file to communicate that to Prolog. Prolog then reads this file, pulls the appropriate information from the relations database and the onology, runs the layout algorithm, and outputs a file of all possible layouts. Lastly, Unity processes and interprets this file, populates the scene accordingly. An illustration of this process is below in Fig 2.

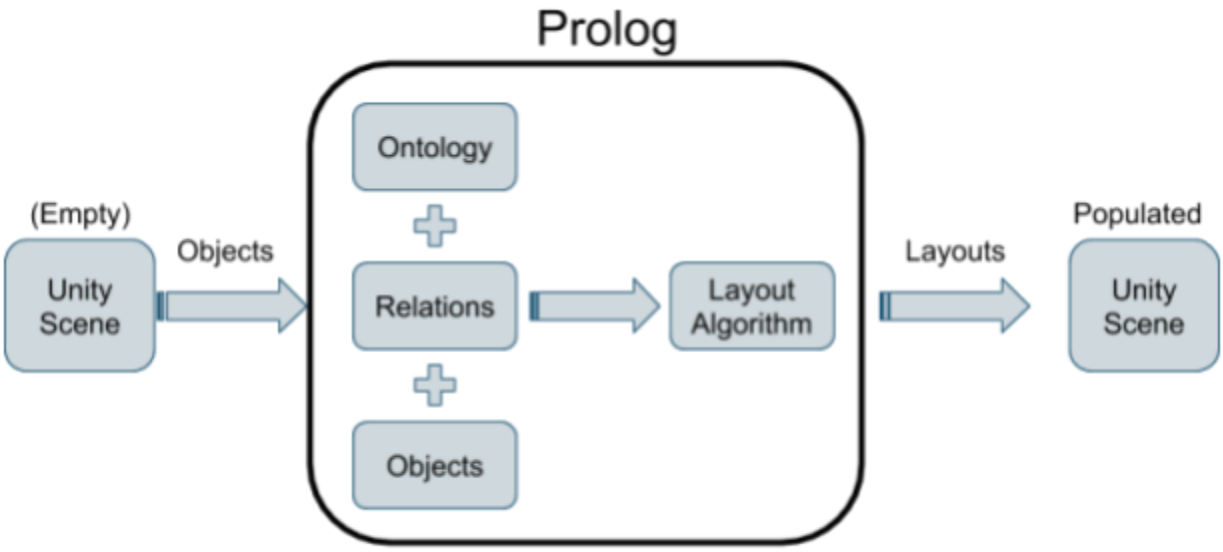
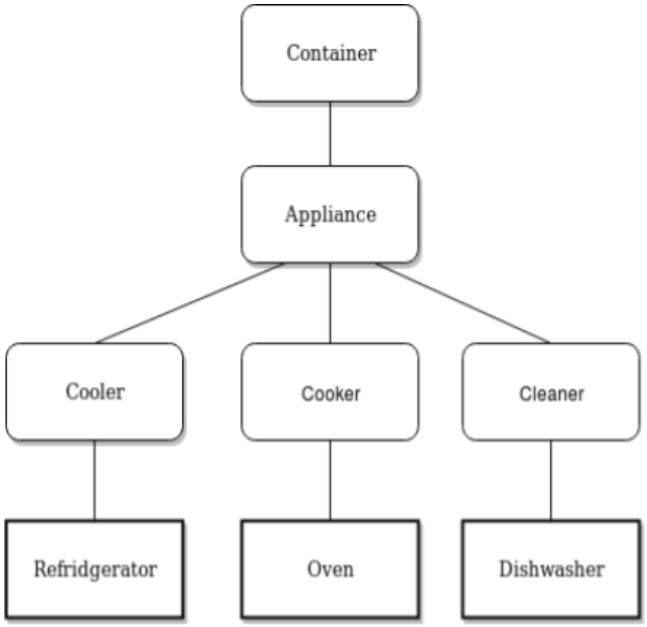


Fig 2: Diagram Illustrating scene generation process.

Ontology

Defines objects and relations (example ontology goes here) Describe these concepts




```prolog
% Container Definition
is_container(X) :-
  is_appliance(X).

%Appliance Definition
is_appliance(X) :-
  appliance(X).

is_appliance(X) :-
  cooler(X).

is_appliance(X) :-
  cooker(X).

is_appliance(X) :-
  cleaner(X).

cooler(refridgerator).
cooker(oven).
Cleaner(dishwasher).
```

Fig x: Shows a basic ontology. On the left is a graph view and on the right is the prolog definition.

An ontology is used to describe hierarchical relationship between objects. This is useful when describing an environment and making a valid or believable relationship. In the above figure you see what this looks like.

The rules that dictate how these entities can relate to each other physically can be looked at as rules describing another valid ontology using this knowledge. This is shown in the figure below.

```
adjacent(X,Y,D) :-
 cabinet(X),
 appliance(Y),
 D \== front,
 D \== behind.

adjacent(X,Y,D) :-
   cabinet(X),
   cabinet(Y),
   D \== front,
   D \== behind.

adjacent(X,Y,D) :-
   wall(X),
   wall(Y),
   D \== front,
   D \== behind.
```

```
in_front(X,Y,D) :-
   cabinet(X),
   wall(Y),
   D = front.

behind(X,Y,D) :-
   appliance(Y),
   wall(X),
   D = behind.

behind(X,Y,D) :-
   cabinet(Y),
   wall(X),
   D = behind.
in_front(X,Y,D) :-
 appliance(X),
 wall(Y),
 D = front.
```

Using this ruleset, we can generate rules with the relationships of X left of Y, Y right of X. Or any direction that is valid. This is then used in order to generate answer sets against an existing list of objects. An example of a list of objects and their output are below.

Virtual Environment

Contains furniture, appliances, tools. Room dimensions. Mapped to the ontology.

These objects are defined as components of a room that should be organized by the layout algorithm. These are both Room and Game objects because they need to be Room objects when the Prolog Layout Algorithm is using them, and Game Objects when Unity is working on them.

Table for names of objects -> count? -> ontology -> rules

Cabinet / wall colors? Or just count

Authoring Tool / Inventory System / Content Specifications

Holds the room/game objects and initial states for room to be generated, and applies the relations between the objects by positioning them in the world. Also specifies the desired content (5 plates, 1 microwave, …) What are all the specifications we can make? Anything about location, grouping? What are these specifications written in? Describe an initial state?

Answer Set is generated by the combination of Layout Solver & Rules Engine. Show an example answer set (small one)

Layout Solver

Takes the required game/room objects (content specifications) and writes their descriptions to a file that can be used by prolog Layout Algorithm in order to generate a valid layout of the objects based off of the defined Layout Ontology.

This is where Prolog is used. What does the output look like? Generates an answer set.

Layout Algorithm takes a given list of objects in a room and constructs an ontology of relationuships between the objects. These relationships are added by the definition of valid relationships defined in the Layout Ontology

Rules Engine

The layout ontology describes a list of rules that should be followed when defining relationships for objects in a room. An example rule would be Food can have the relationship of "in" a Fridge.

The rules for ASP that Prolog follows are defined here. What all rules do we have defined? Justify each. Give an example of a couple rules in prolog and describe them.

Content Mapper / Logic Interpreter / Some good name ...

Takes an answer set (there are several; are they all valid??), parses the answer set, creates instances (in the virtual scene) of each object in the answer set (randomly chooses the concrete "plate" or "cup"), then places them according to the answer set (which is in relation to other objects in the answer set). Please give a short example to illustrate (like placing 3 plates in a cabinet, or a mug on the table, …)

We had two different approaches to implementing this design. The first used Prolog to convert the kitchen space into an n x m grid. Each cell would be marked as occupied or unoccupied, ensuring there existed a valid path between ever occupied cell. Objects would then semi-randomly populate the occupied cells. This approach was influenced by the maze generation done by Smith.[1] This approach created two major problems. The first problem was that the complexity increased exponentially with the size of the room and the number of objects being added. This meant that the algorithm would only be appropriate for smaller rooms with few objects.The second problem was that the algorithm was unable to account for objects that were larger than one cell. An object may take up multiple cells in the scene and correspond to the cell dimensions. This caused a false sense of validity, as the objects would not be set in the appropriate positions relative to their sizes.

After struggling to solve these problems, we decided a major shift in ideology was required. Our new approach defines objects relative to each other rather than using absolute positioning. This takes away the concern for object size in Prolog and shifts the burden onto Unity which handles object dimensions well. Prolog now solely creates sets of layouts in the form of object-connection pairs. This is a more appropriate use of Prolog than our previous method which attempted to functionalize a declarative language. The new approach works

well with the ontology in defining layouts which provides a powerful tool for feeding information into AI systems.

The new approach solved the first problem of the exponential complexity with respect to the room dimensions and the number of objects. Although the collision problem was not solved, handling the problem shifted over to Unity which is much better at collision handling. This means that collisions can occur, as illustrated below in Figure 3, but that Unity can now detect any collisions and adjust the room or objects accordingly. To solve the collision problem, a simple fix was employed. When collision occurs, shift all objects along the wall until all the object collides no longer. This is a temporary fix as it will not work in all scenarios and not ideal for future work.

Fig 3: Showing the clipping issues that could occur without collision detection.

2.3. Tasks

Emmanuel:

- Program C# scripts to read in and interpret prolog files
- Build new scenes within Unity using the C# scripts
- Validate proper item placement and scene generation

Scott:

- Declare prolog rules that need to be satisfied including
  - Relation Rules
  - Object Types
  - Placement Algorithm
- Incorporated Ontology into prolog rules

Preston:

- Pre-process assets to be used in generation

- Set Interactable & Inventory Locations
- Object Definitions
- State Collections
- Reaction Collections
- Animations

4. Testing

3.1. Methodology

When testing our first approach, we started by implementing the Chromatic Maze[1]. This test was designed to gauge the feasibility of using the grid-based format. The maze test used Unity as its output and was marginally successful. This proved that Unity is a viable program for displaying the layouts.

From this point, we used the code from the chromatic maze test to show when a cell was occupied by an object. This highlighted two problem we did not anticipate. With the grid approach, the objects could be smaller, or larger than the grid spaces. Even if the cell sizes were modified to fit the specific shape and size of each object, the complexity of the problem slowed Prolog down greatly. On small 4x4 grids with 3-4 objects, it would generate layouts almost instantly, but when shifted to 6x6 grids with 7 objects, it could take several minutes. While this doesn't sound like a long time, the anticipated number of objects and room dimensions are much larger. A test of this algorithm is shown in Figure 4 below.

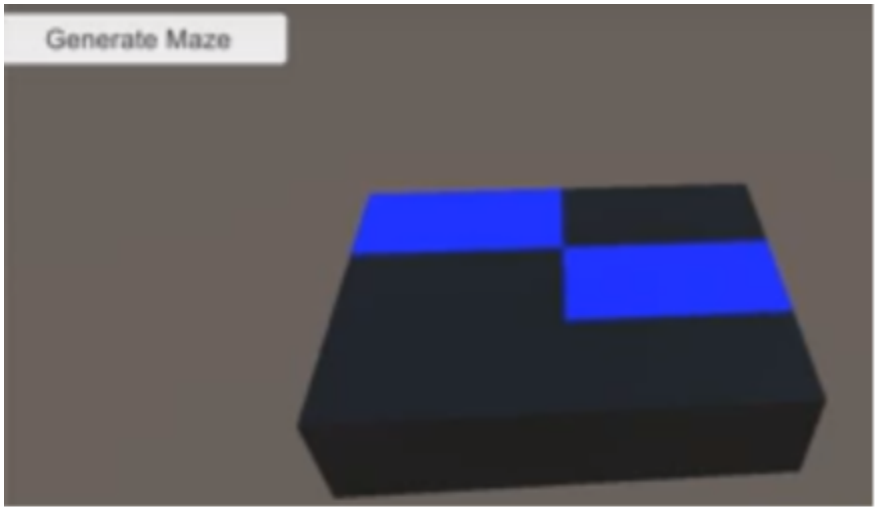


Fig 4: Generating layout from grid space with pathing algorithm.

With this in mind, the pathing algorithm was dropped, and now the relationships between the objects are all that is computed. Now the answer sets no longer have to be concerned about positioning, only relationships. How this is designed is that certain types of objects are capable of sharing a relationship with one another, and incapable of doing the same with others through the lack of specification. These sets of rules, or valid relationships, define what is possible in the layouts.

What this means, is that Unity must take care of the potential collisions with other objects, including walls. Once a basic ontology was defined, we created the new interface between SWI-Prolog and Unity to send the objects in the room, and then receive the layouts. This was successful, where the room can generate simple ontologies of a room. It was missing a proper connection to the room. For this reason, the room anchor was made. The room anchor is the part of the room a group of objects are connected to. This could be a wall object or floor tile. These anchors are the position that all the objects will be connected to and positioned relative to. Currently, room anchors are selected randomly from all options, and are then used and defined in the ontology. These anchors are the first connection and “root node” of the room to the rest of the objects grouped together.

3.2. Results and Analysis

From the initial tests, the Unity side needed some work on detecting correct collisions as illustrated in Fig 3.

Afterwards, the collision detection was adjusted to check for multiple frames. The result is now layouts that are generated now are shifted until they are able to fit inside the room. It isn’t perfect but works most of the time. Future work should tackle this problem more in depth, as the problem will grow in complexity as the project moves away from the prototypes.

5. Conclusions

4.1. Summary

The start of this project had many goals and milestones that were not reached. The biggest reason for this was Prolog. It is the right tool for the job, but the technology was foreign to the team and it took a quarter to realize that the approach being used with the grid space was inappropriate for Prolog. Once the new design was made and incorporated, it was a natural fit.

Prolog is great for defining room layouts, and perhaps more, but many other components of PCG should be employed by other algorithms. This project has given some insight on where to use other PCG techniques, such as binary partition trees to partition space and floor plans for rooms and corridors.

4.2. Future Work

There are many aspects of the project that can be improved upon. Our project relied upon a crudely created ontology with a hierarchy that only applies to a simple kitchen scene. In the future, utilizing already existing ontologies via OWL would be beneficial. If there are no ontologies related to a scenario that is needed, the ontologies built for that scenario should be contributed.

With the kitchen completed, the same methodology can be used to apply the algorithm to various other environments on a larger scale. For example, generating a layout for the whole house, a hospital, and even a city.

Currently for prototyping, the ontology definitions are overlapping on the Unity side and Prolog side. Needing the objects to know what time of objects they are from within Unity causes redundancy and version control issues. It also relies on the person creating and populating the objects to have extensive knowledge of the ontology. Having the ontology solely defined in Prolog is the long term solution, as the definitions should only happen once, and updated in one location for future changes. This was done for prototyping purposes as the Unity side needed to use fake ontologies for testing before Prolog was running.

A naming convention should be established for projects that wish to use this tool, as the names of objects are currently what is used to connect the layouts generated by Prolog and the Unity gameobjects.

Creating an ontology editor inside of Unity would help speed up the process of integrating an ontology into the project. This could be used also to add properties to objects, and have a live view of what potential layouts would look like.

Creating a system to log changes in the dynamic ontology would be a great debugging tool when using the generated environment to execute the workflow, and for future analysis in machine learning models. There is a need to log these dynamic and static ontologies.

Storing the ontologies would get messy as a filesystem. It would either turn into one big file, or a collection of them. Storing them inside of a database may be ideal, and should be tested.

This would also be a great place to store the dynamic ontology as the workflow takes place. A filesystem may be easier to integrate into existing projects.

Current semantic image segmentation tools rely on layers in the unity scene to define how objects should be grouped together. It would be a lot more natural to segment them into the already defined object classes inside the ontology, and much more flexible, as to change the image segmentation object focus, one would not have to edit the layers of the scene but the list of object categories to segment.

4.3. Project Value

This projected was selected because the thought of being able to generalize procedural content generation was an intriguing challenge. We were hoping to make a lot more progress, but then again, we are using a technique currently not employed beyond the academic testing sphere, and creating the foundation to the generator from scratch. We learned a lot of where and when to use this PCG technique, and are excited to try more and to flesh out more features for the one we made.

Bios

- **Dr. Wesley Deneke, Mentor** – Deneke is a professor in the Computer Science Department. He leads student research projects that are currently focusing on how to simulate Human Workflows using 3D virtual worlds.
- **Emmanuel Harley** - Harley is a student at WWU studying Computer Science. His main work has been on Unity for this project, and connecting prolog to it.
- **Jon S. Spitzer** - Jon is an alumnus of WWU that graduated with a degree in Computer Science. He worked primarily on the Prolog side of the project, developing the ontology, relations, and layout algorithm.
- **Preston D. Carroll** - Preston is an alumnus of WWU who graduated with a Computer Science degree. His work has mainly been in Unity, preparing assets to be used in content generation.

References

[1] A. M. Smith and M. Mateas, "Answer Set Programming for Procedural Content Generation: A Design Space Approach," in *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 3, no. 3, pp. 187-200, Sept. 2011. doi: 10.1109/TCIAIG.2011.2158545

[2] J. Xu, D. Vazquez, A. Lopez, J. Marin, and D. Ponsa, “Learning a Part-Based Pedestrian Detector in a Virtual World,” IEEE Transactions on Intelligent Transportation Systems, vol. 15, pp. 2121-2131, Oct 2014.

[3] H. Hattori, V. Boddeti, K. Kitani, and T. Kanade, “Learning Scene-Specific Pedestrian Detectors without Real Data,” IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pp. 3819-3827, 2015.

[4] C. Moate, et al., “Vehicle Detection in Infrared Imagery Using Neural Networks with Synthetic Training Data,” International Conference Image Analysis and Recognition, pp. 453-461, June 2018.

[5] C. Chen, A. Seff, A. Kornhauser, and J. Xiao, “DeepDriving: Learning Affordance for Direct Perception in Autonomous Driving,” IEEE International Conference on Computer Vision (ICCV), pp. 2722-2730, 2015.

[6] D. Vázquez, A. López, J. Marín, D. Ponsa, and D. Gerónimo, “Virtual and Real World Adaptation for Pedestrian Detection,” IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 36, pp. 797-809, Apr 2014.

[7] H. Alhaija, S. Mustikovela, L. Mescheder, A. Geiger, and C. Rother, “Augmented Reality Meets Computer Vision: Efficient Data Generation for Urban Driving Scenes,” International Journal of Computer Vision, vol. 126, pp. 961-972, Sep 2018.

[8] Puig, et al. “VirtualHome: Simulating Household Activities via Programs.” [Astro-Ph/0005112] A Determination of the Hubble Constant from Cepheid Distances and a Model of the Local Peculiar Velocity Field, American Physical Society, 19 June 2018, arxiv.org/abs/1806.07011.