MALMÖ UNIVERSITET

Computer Science
Faculty of Technology and Society

Hierarchical Procedural Decoration of Game Environments

Daniel Borgshammar

Ola Lidberg

Thesis: Bachelor 180 credits
Main Field: Computer Science
Program: Game Development
Date of final seminar: 2020-06-01
Supervisor: Steve Dahlskog
Examiner: José Maria Font Fernandez

Hierarchical Procedural Decoration of Game Environments

Bachelor thesis in Game Development Studies

Daniel Borgshammar, Ola Lidberg
Department of Computer Science and Media Technology
Faculty of Technology and Society
Malmö University

June 2020

Abstract

Immersive environments are a big part of video games. With hardware being more capable than ever of displaying millions of triangles at once both the capability and demand of more content in an environment increases. This puts more strain on level designers who have to spend more time per scene to make everything meet the constantly increasing visual standards. Being able to procedurally decorate any environment in such a fashion that it follows the hand-made hero scenes of the game becomes an important way to save time and money better spent elsewhere in the project.

This thesis investigates a hierarchical method of procedurally generating objects inside an environment. This was accomplished by creating an artifact to analyze the feasibility of efficiently hierarchically decorating environments at run-time of a video game. Though focusing on furniture and decorative objects of rooms the system can also be used to procedurally place game related content.

1 Introduction

Level Design is an increasingly complex task as game worlds grow more and more elaborate. Time required for basic tasks such as constructing buildings and decorating the interior grows exponentially with the game complexity and the predicted demands of its future players. [1] This creates an increased demand for procedural generation of world content to allow the designers to focus on more important tasks. [2] World content needs a certain level of believeability. Procedurally generated content often appears too chaotic, or too deterministic, and causes a loss of connection with the world and/or loss of fidelity. It's also time consuming to fix and touch up after generating, which can take almost as long as manually building it. [3]

Procedural content generation is becoming an increasingly popular method for adding extensive content to modern video games, providing an effective means of saving time for many different areas of the development team. [4] While it has already been used extensively to provide extended areas for the player to explore these are often limited to landscapes, cityscapes, cave systems, star systems or other large and relatively simple vistas. Additional uses keep being thought of such as procedural animation, creation of characters, particle systems and even whole levels depending on the type of game. Many times it saves time for the level designers and environment creators but the more overarching the task of the procedural generation system the more human involvement may be required to prune results that are not useful. Ideally environments that can be constructed procedurally should not require user intervention but should look convincing enough that a designer may have created it by hand. [5]

Many games provide elaborate cities with a plethora of buildings which help provide valuable story-telling mechanics by setting the look and feel for the environment the player explores. Time, budget and/or technology constraints often cause only a handful of buildings to be available for exploration and those which are often tend to have many closed off areas where the designers have not placed any content. [3] While it could be argued that not every room is available for exploration if you enter a building, and not every room may be relevant to the game, it does take away some of the freedom of the player and some of the depth of the game world. [5]

Effective examples of this includes "Assassins Creed: Syndicate" (2015) and "The Division 2" (2019). Syndicate offers a series of buildings which can be entered and run through in order to escape some chase sequences or to hide from enemy characters in the game and while they are decorated they tend to only provide two-three rooms in order to facilitate a corridor leading from one side of the building to another. [5] Meanwhile Division 2 offers a small handful of buildings which often take the shape of a "Dungeon" with stacked barriers preventing you from exploring the entirety, to small sequences of apartments that act as a method of moving between ground level and rooftops. While both games offer more exploration of buildings than their predecessors they are still heavily limited which leads to less of an impact of their game worlds. This is especially present in The Division 2 where it often leads to a feeling of strict corridors as you explore the disaster-ridden city of Washington D.C. In the wake of the pandemic that plunged the world into disaster the player often draws parallels with open-world exploration games and movies and a strong desire to explore abandoned homes and other buildings to find resources is let down by the limitations of the game world.

Being able to procedurally decorate rooms, apartments or even whole houses with both furniture and game-relevant items would both help avoid these limitations and encourage players to take deeper part of the game world. [5] Our work primarily aims to provide a system for furnishing rooms in a real-time, procedural manner that offers a consistent and semi-predictable result by means of seed-based hierarchical decoration. This allows creation of large amounts of furnished buildings on the fly and helps reinforce the illusion of the world that you are exploring. Buildings can be created as empty shells which are then populated by floors as the player gets closer, and finally furnished rooms as they get near or opt to enter the building to explore. Likewise the building contents can be unloaded when no longer needed to keep the memory available for the next building the player may enter.

The system needs to be able to create believable solutions for furnishing a room where no two rooms end up looking exactly the same–how often does one see living apartment buildings where every apartment looks like it came off of an assembly line? Indeed this behavior is often just limited to hotels, and even those rooms end up looking individual after a guest has checked in. However, rooms must also be able to be represented by a numerical value so that it can be reconstructed when needed in the same way it appeared originally–a room is seldom different when you return to it after fetching a cup of coffee.

With focus on furnishing we have established a hierarchical system which, using a semantic data table as backing store, furnishes a room on the fly while also decorating it with relevant objects or game-related items.

2 Related research

With the steady increase in processing power video game environments end up being more complicated with more objects decorating the environments (Figure 1). This puts higher demands on level designers to assemble a scene which is aesthetically pleasing and mechanically functional to meet the requirements of players. Time is a major constraint for these tasks and being able to save time by procedurally generating content would allow the designer to focus on other things.

Research on procedural generation of content has focused on different areas. Some have supplied shape grammars based on the decomposition of the hierarchical structures of for example buildings and floor-plans. See for example the work of Wonka, et al. (2003) [8] and Müller et al. (2003) [9].

Another approach of floor-plan generating was presented by Hahn et al. (2006) [10]. They split the floor to generate building interiors at random positions and connect the resulting regions by por-

Figure 1: Comparison of video game fidelity evolution. "Doom" [6] (left) and "StarCitizen" [7] (right)

tals. To make interactive walk-throughs possible, they used a lazy generation scheme together with a persistent change manager. This was useful for generating floor-plans, however it did not take into account the furnishing of the interior.

Although many researchers have done work in the field, only a few have focused on generating interior and decorations. For example, in the work of Tutenel et al. (2009) [5] they introduce a novel rule-based layout solving approach. They try to use this solving approach by providing the solver with a user-defined plan for procedural generation. In this plan, users can specify objects to be placed as instances of classes, which in turn contain rules about how instances should be placed.

Germer et al. (2008) [11] implements a system to generate furniture arrangements on the fly while the user explores the scene. The authors use an agent-based approach where every single piece of furniture are an autonomous agent that are able to move around and arrange themselves properly. According to their own evaluation of the system it has some drawbacks. There is for example no way of telling a TV to be placed opposite of the couch, the system does not prevent objects from blocking each other and if the user rushes through the rooms the system will not be able to generate enough rooms.

The purpose of this study is to implement a hierarchical system that generates believable environments which do not appear to be computer generated and decorate them on the fly.

3 Method

This research project has used a Design Science Research Methodology (DSRM) approach as described by *Peffers et al. (2006)* [12]. In their paper they motivate, present, demonstrate in use, and evaluates a methodology for conducting design science (DS) research in information systems (IS). DS is of importance in a discipline oriented to the creation of successful artifacts. The design science research methodology (DSRM) presented by the authors incorporates principles, practices, and procedures required to carry out such research and meets three objectives: it is consistent with prior literature, it provides a nominal process model for doing DS research, and it provides a mental model for presenting and evaluating DS research in IS. The DS process includes six steps: problem identification and motivation, definition of the objectives for a solution, design and development, demonstration, evaluation, and communication. See Figure 2.

3.1 Defining the Problem

Our problem is two-fold; how do you efficiently decorate any space in a hierarchical manner, and how do you do it in a way that makes it appear as if it was not computer generated? Procedural content generation for video games has existed for some time now and range from particles and animations to entire planets–nearly any field within game development has some form of procedural technology available. To make procedural content appear human generated and not have a blatant cookie-cutter presentation one must delve deeper

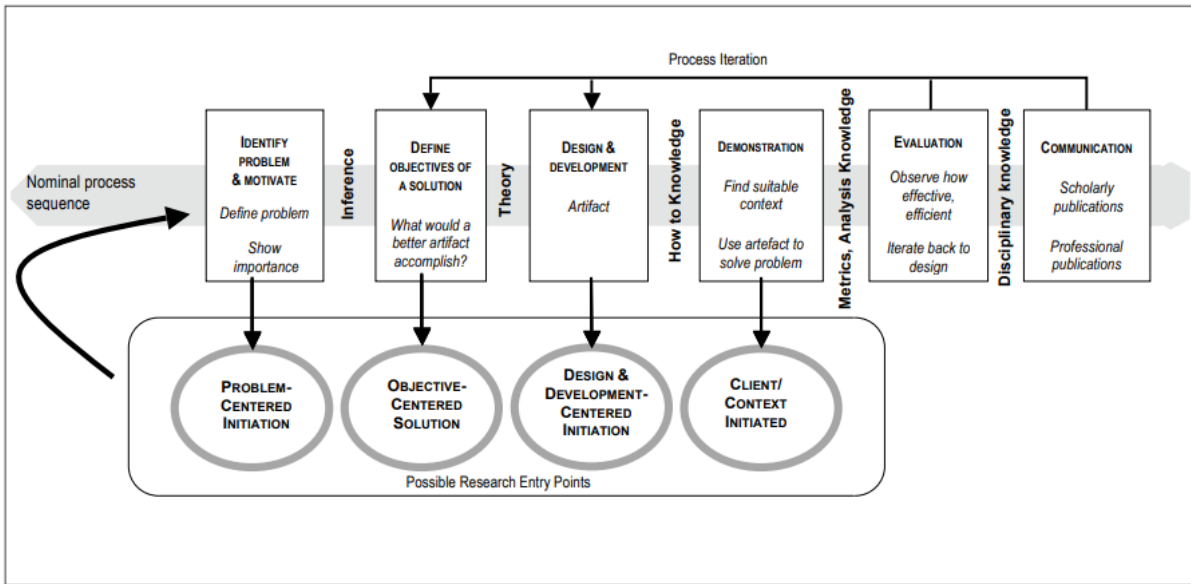

Figure 2: DSRM, Research Process Diagram [12]

into what makes things look organic and natural. In our case it is the investigation into what makes a space appear human generated and actually lived in. This can be extremely difficult to test using any algorithm; a genetic algorithm could eventually be taught what constitutes organic and natural in space composition but this would undoubtedly take an extended amount of time and pose a problem which is part of the definition of what we have here; it would still appear computer generated to some degree. Therefore the only viable analyst in this circumstance is actual people.
But what constitutes a realistic space? Obvious guidelines include not having furniture placed in peculiar ways such as stacked in a corner, or hanging from the ceiling, although one might argue that stacked in a corner is a viable organic indicator while storing furniture, requiring space and so on. Likewise one would not want furniture that is aligned with millimeter precision in every instance, though there are people who prefer it this way as well. Therefore we have defined a reasonably realistic appearance of the environment.

3.2 Goals

Our primary goal is to define a system for hierarchical procedural decoration of an environment and for this environment to appear as organically designed as possible; an environment which looks like it was populated by a designer and not a computer algorithm. The system needs to be able to fill the designated space with items which may in turn define other items, both content and children, in a hierarchical manner by having objects depend on one another up the chain. The system should preferably be capable of doing this in real time so that the contents may be populated and scrapped as needed while still being able to restore an already generated room if needed by the game environment. Further the system has to offer flexibility and ease of use despite the apparent loops which hierarchical systems can exhibit:

- A real-time hierarchical decorative system capable of furnishing rooms, apartments, buildings or even entire cities on the fly.

- Furniture has predetermined types and rules for children that they can either want or require, which are then created once the parent is satisfied with its position.

- Children can request that the parent makes room for them if possible, depending on the provided rules.

3.3 Artifact

In order to facilitate the intended decoration system we will design and implement a procedural hierarchical decoration system in Unreal Engine 4 [13]. The primary implementation will be done utilizing the proprietary visual coding system Blueprint to facilitate quick iterations and easy debugging of the execution flow without having to recompile any source code. Unreal Engine 4 provides a ready, modern and popular game engine which provides a good testing ground for the project. It also allows a quick and easy implementation of a basic game system to further test the hierarchical decoration system and it can be easily packaged into a ready to use component which can be easily added to other projects as needed.

3.4 Evaluation and Data Collection

To define a baseline for what constitutes a believable environment we intend to analyze a series of images, both photographic and CGI, to build a useful understanding of how spaces are made to appear artificial and not without intending them to be such. In turn the decorated environment needs to exhibit the same qualities identified in this study to offer a believable layout that does not appear as if something mechanical has put the contents of the space in arbitrary places.

For evaluating the generated environments a testing group needs to rate the environments based on how natural they look. We plan to do this using a series of generated images in combination with a manually decorated one, and a questionnaire where participants can look at the images and rank them on a scale as well as a free form text field which allows individual feedback on each item. This will help build an understanding of what people perceive as realistic and gather important feedback. It is important that the testers have no prior information about the images before they look at them. The participant should spend a few moments looking at the images and then move on to the questions. It is our understanding that the believability of a space is determined subconsciously–most people do not consciously think about the space they enter as they do so unless there is something profoundly out of the ordinary.

4 Procedural Hierarchical Decoration

Our method for decorating spaces offers a way for environment designers to specify key areas of an environment which needs to be decorated in a certain fashion, using certain rules, by employing Procedural Decoration Volumes to do this task. The designer has full control over the settings of the decoration system at design time and may specify either a theme for the entire environment or on a per-volume basis. Volumes can be labelled as different kinds of areas by the use of tags and read object information from a database object, including positional and spawning semantics. Each object can also reference back to the same database when defining child or leaf objects. Object actors can in turn contain their own volumes which will be procedurally filled in turn after the object has been successfully placed.

4.1 Procedural Decoration Volume

A Procedural Decoration Volume, or PDV, is a rectangular area which defines the space the decoration system should fill with items when requested. These areas can be of any size and may vary from a small portion of a larger environment up to and including the entire area such as a room. Upon creation the PDV will start by partitioning the space available inside it into smaller cubes, or cells, given a ratio per meter. These cells represent the state of space at their location and may have three potential states; Unoccupied, Tentative and Occupied. Unoccupied cells have yet to be filled with anything and are available to be claimed by objects which are placed in the volume. This is the default state of all cells on creation. Tentative cells are those which have been searched and found empty when an object is looking for available space around itself to place a leaf object. Occupied cells have been claimed by a placed object and cannot hold anything else.

Once the partitioning is done the PDV will optionally perform a series of line traces near its edges to determine if there is empty space or not. Traces will be fired at certain interval distances and at both the bottom and middle of the volume. For rooms this allows the system to find walls, windows and doors which should be treated as already used space so that objects will not placed to block them off. Windows and Doors are treated differently, allowing furniture under a certain height to be placed under windows but not in front of doors.

Next the PDV will inspect its content rules and gather information about all possible objects that could appear within its bounds. Each object has an environment score based on a floating point value between 0-1 and a Type tag. Objects with a value of 1 must be present in the particular area while objects with a score of 0 act as if they are not present at all and will not appear. Those objects which have a value of $0 < n < 1$ may appear if there's room for them to do so and if they are selected using which ever selection routine the designer has specified. Each PDV may also define rules that dictate how many of these required objects it needs to have spawned to be considered valid and, if it fails to do so, will signal that the object it belongs to has failed to populate successfully. In most cases this will cause the parent PDV to select a new object and try again while root PDVs such as rooms will simply log an error and give up. Similarly objects will be filtered on their Type tag to prevent, for example, furniture from appearing on top of desks and in bookshelves as well as small decorative objects being placed on the floor.

4.2 Rectangle Model

Like the PDVs all objects are approximated down to bounding rectangles regardless of shape. We found this to be a good representation of both most pieces of furniture as well as most decorative objects. This allows an efficient and fast method for calculating the amount of space an object requires and scanning for available space in the PDV. It also simplifies determining the positional constraints of the various objects–a rectangle can have a front, back, left and right side given a forward axis with ease. Objects can then define other objects which they would like to have next to themselves in any of the four cardinal directions and then define their own PDVs for objects that they would like to have appear on top or beneath, on available shelves and so on. This system allows, for example, a bookshelf to spawn a decorative plant next to itself and then have a PDV for each shelf that then populates with books or other objects.

4.3 Previous Attempts

During our research on the subject we went through quite a lot of different variations of the decoration semantics. Our initial method involved not using subdivisions at all but instead floating rectangles in a volume. This method still harnesses the rectangle model but relied on overlap checks to determine if an object was already located in a certain spot and, if it was, keep moving our object by an arbitrary distance until it no longer overlapped. Though efficient in theory it has several shortcomings such as making it impossible to estimate how full a room is, making it difficult to easily move furniture to make space for requested children and having difficulties determining if objects can fit where they are supposed to.

Another method employed the subdivisions on a smaller scale where objects were spawned in the middle of the room, then pushed until they hit a wall at which point the cells would be flagged as occupied. If another object was hit it would then move sideways before attempting to continue going towards the wall. This was just not efficient or feasible, not to mention reminiscent of a Russian block game rather than decorative furnishing.

4.4 Decoration

Each object has its own placement rules which define where an object may exist within the volume, how much space it would like to have around itself if possible, which items it may spawn around itself if any, and if the object is a leaf-only object and as such can only exist if requested by a parent object. Taking these rules into account the PDV will attempt to place items according to the selection rules provided by the designer.

When selected an object is first measured in spatial cells to establish a footprint size. The decorator will find a location which appears to satisfy the positioning requirement of the object and then check to see if the footprint will fit in that location. If

it does, the object will be placed at the center of this footprint and otherwise another position will be selected.

Once successfully positioned the object will check if it has any child objects to create. If there is at least one child leaf the object will scan to see how much space it has available around itself in each direction. Leaf objects can exist to the left, right, in front of, behind, above or below their parent. This allows, for example, a computer desk to spawn an office chair in front of itself and a small chest of drawers next to itself as needed. If the leaf object cannot fit in the place it would like to be it will ask its parent to move, if possible, to make room for the leaf. This move operation may cause a call back up the hierarchical chain until requested space is found or a parent is unable to make room to accommodate the leaf object.

Parents are also capable of requiring certain leaf objects and may flag themselves as invalid if these fail to spawn. Invalid objects are removed, and another object is selected in its place. Leaves are placed facing their parent by default and can have a slight deviation to add some more believeability to the environment so that everything doesn't appear statically positioned. Once all leaf conditions are satisfied the parent object will enumerate any PDVs that it itself contains and the decorator will move on to working on those in turn, stepping down a level until no more PDVs are present, establishing the other half of the hierarchical decoration.

4.5 Environments

Each PDV has a setting which controls the type of environment that it represents and this effects the types of content that it will generate. If these settings are not specified it will try to find the next PDV in its chain of parents until it finds one with its environment set, or error out if it cannot determine what to do. We found this preferable to just choosing objects from all environments and to help guide the designer about the intended use of the system.

An object may have more than one environment defined and can have different probability scores for each type of environment. As mentioned before objects with a score of 0 are treated as if they're not present at all, and objects with a score of 1 are treated as required in that environment. This allows you to, for example, require a toilet and bath tub in a bathroom but a shower might be optional. Likewise a kitchen may require a stove but a dishwasher is optional. Both types of rooms may share a certain type of ceiling light which has a chance of appearing in both environments.

In addition to this an object may specify a range of how many times it would be likely to appear in a given area. You may have more than one bookshelf in the living room or office, and more than one sitting room chair and so on. The system will endeavor to create a number of these objects between the provided minimum and maximum and if it fails to spawn at least the minimum amount of them it can optionally fail silently or invalidate its parent depending on user preferences.

4.6 User Input

For our system to work it requires a certain amount of user input prior to running. This section is intended to explain first the minimum user input required to operate and then offer an overview of extended configuration options available to help with the environment generation. The bare minimum settings required are:

- Each root PDV (I.e. room, environment, etc) must have an Environment Type set.
- Each root PDV must have a Content Type set.
- A source of objects for the PDV to choose from must exist and be assigned to each root PDV.
- The source must contain the following information:
  - Placement semantic settings.
  - Object type and
  - Object Class.

Environment Types define the general type of content that will appear in the PDV. These are overarching labels such as room types; Livingroom, Bedroom, Home Office, Kitchen, Bathroom and so on. It tells the PDV which specific environment to match on when gathering potential objects during initialization.

Content Types denote the object type itself such as Furniture, Essentials (for example a computer

for a office desk, books for a bookshelf, etc) or Decorations (potted plants, lamps, sculptures, etc). This setting is used to help a PDV determine what the user expects it to generate, preventing furniture from spawning onto of a desk, or books from being lined up on the floor.

The placement semantic settings dictate how this object should be placed. It determines if the object should stand against a wall, at some arbitrary spot in the room, which direction it should face relative to its position and any padding that it might want around itself at all times. It also dictates if the object is unique or can only be a leaf. Unique objects can only appear once per room, such as a stove, shower or toilet while leaf objects are required to be spawned by their parent. These rules also contain the space reservation mode, explained later, and any leaf requirements the object may have, that is objects that it can spawn next to itself such as a table expecting a chair next to itself. Finally it also contains a list of potential environments that the object may appear in, and the likelihood of it doing so as a percentile float value.

Using these basic settings the system will be capable of filling any volume with objects automatically within its default limits but it will not be particularly interesting. By default a PDV will only spawn required objects once and not spawn any optional objects at all, expecting the user to configure the requirements for these. The same goes with the max hierarchical depth which defaults to 3. This provides a good default starting point, preventing objects from diving too deep while demonstrating how the system works in a pleasant manner.

4.7 Persistence

If the viewer moves far enough away from a procedurally decorated environment that it must be destroyed the room must look the same when the viewer comes back to it. To account for this a PDV that has already laid out once will save the random number sequence it used to initialize its random number generator. This guarantees that the environment will be generated with exactly the same appearance on each call.

Of course this does not account for objects which may have been moved or collected by the user but this could easily be accounted for by storing a small data structure for each generated object. This is much more efficient than persisting the room itself with all the models and textures that it requires and can be stored indefinitely for the duration of the game if required. Often the restoration of rooms to their original state is acceptable within the game world if you travel far away enough first and such behavior may even be expected by the player. In such circumstances it could be preferable to just track objects which matter such as game pickups so that they are not repopulated, or even have them generate a different pickup every time to allow some variation.

4.8 Algorithm

Our algorithm for procedural hierarchical decoration goes through six distinct phases during the decoration process (Figure 3). These phases have two locations where they can step down on two different hierarchical domains; object and PDV. The object domains hierarchy is defined by parent-leaf relationships between objects and may span down to the defined maximum depth of the PDV. Deeper levels allow for greater complexity in the scene but also increases computation time and probability of an unsolvable environment. The PDV domains hierarchy is defined by parent-child relationships between PDVs. A PDV may spawn an object which in turn contains one or more PDVs which then become children of the parent PDV and inherit its object source and environment rules. The reason for these not being considered the same domain is because the contents of an object, such as items on a table or in a bookshelf, are expected to solve their own layout unaffected by the parent–the PDV is only aware of its own space and operates therein. This being said both domains hold references to the other to maintain an interlinked hierarchy. Though it may sound complicated care has been taken to make sure that this linking only occurs at levels where it makes sense; PDVs may invalidate their parent object when requested, potentially causing the object to make a call to its parent PDV that it has failed its validation rules. In turn the parent PDV may fail validation and request that a child object removes itself, its leaves and any PDVs it contains. Other than these specific circumstances the two hierarchies do not affect one another. Invalidated PDVs will hierarchically clean up any child objects and their contents and PDVs when requested, en-

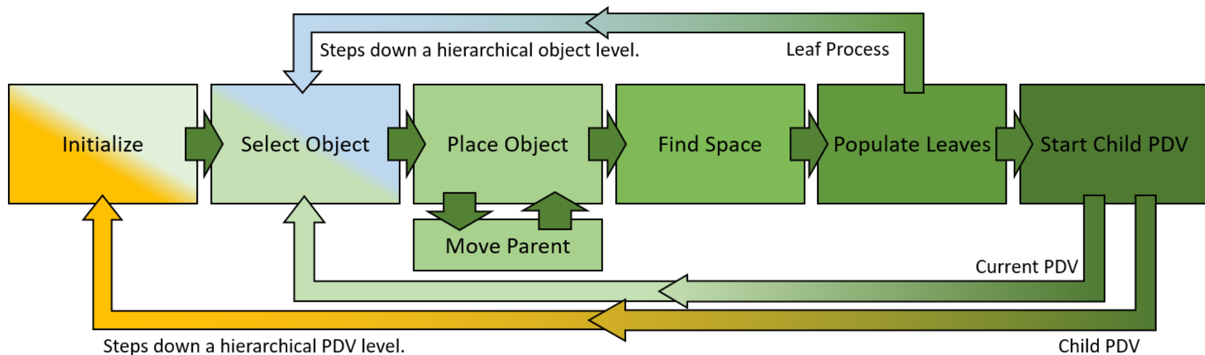

Figure 3: Procedural Decoration Volume Process Diagram

suring that no errant objects are left behind.

Initialize The PDV begins by subdividing its volume according to the requested granularity and creating a partition map which holds the cell structures with information about its subdivisions. Next it will search for its parent environment and any possible direct parent PDVs and ask them for an object data source, the environment type and the randomized environment score. It may also ask the parent environment for a random seed or stream to use when decorating, but this is an optional feature. When subdividing the PDV also sets up "PDV Space", a spatial representation of the volume itself with origin in the rear-left corner (-X, -Y in Unreal Engine's 3D space, at 0 rotation and translation). All subsequent operations take place in PDV Space to make calculations easier by avoiding potentially negative coordinates. This means that PDV Space goes from 0,0 in the rear-left corner to Volume Extents X and Y * 2 in the front-right corner. After this the PDV will gather all required and optional objects that it can contain by going through its data source and matching with its content type and the environment score. Required objects, that is objects with a matching environment type score of 1.0, are stored in a separate acceleration map from optional objects, those that have a matching score of 0.01 to 0.99. If the PDV is a root PDV, that is it is owned by a Room parent, it will scan along every edge cell to check if there are holes in the environment walls such as windows or doorways. If any are found they will be flagged as Occupied in the partition map with the requested depth, a range in cells going from the edge towards the center of the room. Finally if everything went well, the PDV will flag itself CanDecorate and wait to be started and, once this occurs, it will continue by calling Select Object.

Select Object copies its Required Objects map to a temporary structure and then enter a loop that runs until this temporary structure is out of entries, or the required number of required objects have been selected. Ths reason for this temporary structure is to keep track of gathered object entries regardless of what population rules the PDV has been given; objects may be allowed to spawn only once if they are unique or PDV settings demand it, for example. Next it will select a random object from its acceleration map by fetching the next integer from its seeded random stream and hand it over to Place Object. If Place Object succeeds the next object will be tried until all required objects have been spawned. Should placement of a required object fail, Select Object will invalidate the PDV and raise an error unless the requested number of required items have already been spawned. If nothing goes wrong it will then repeat this entire process with its list of optional objects.

Place Object is given an object definition from which it selects a random variation, and uses this variation to create the requested object in a neutral position, ignoring any existing collisions and without updating the partition map. The object is then measured to establish the footprint size in partition cells by measuring the bounds of the object, including any padding as required by the object settings, and a starting position for where to try to place the object is calculated. The location and method

of finding this position varies with the positioning requirements of the object and if the function was called with a parent object or not. If the object has requested to be positioned against an edge (a wall for example), an edge will be selected at random and a shuffled array of all possible cell positions along that edge will be created. These positions will then be tried in order by Find Space. If none of the positions can satisfy the footprint the next edge will be tried in turn. Each time an edge fails to offer space it is removed from subsequent attempts by this call to Place Object. If instead the object has been requested to be placed floating in the room the function will select a number of cells from a subset of those found in the partition map. This subset is calculated by removing all cells that are within the padding range of potential doorways and windows, essentially subtracting a border of cells from the PDV partition map which it cannot choose from. These positions are then handed over to Find Space just as those generated by the edge placing algorithm. When provided with a parent object the function will ignore the first two methods of placing objects and defer the location to the parent-child relation of the object. The parent defines which edge the object should be placed against, if the object should be placed in-line or centered along that edge, and if it should face the parent, face away, or inherit the parents rotation. Place Object will determine the starting location for Find Space based on these rules. If Find Space fails to reserve a place for the footprint Place Object has generated it will check to see if the objects being in the way are either our parent or a sibling and, if this is the case, ask our parent to move to make room by calling Move Parent. If the object and all its leaves (and their leaves, respectively) were successfully spawned the PDV will continue with Populate Leaves. If something went wrong it will check if its object definition has other variations of the object and try one of them until they have been exhausted, returning an error to Select Object, or until a variation is successfully placed.

Move Parent will create a list of its own footprint and that of all its already placed leaves and then offset these footprints in the requested direction by the requested amount. Next it will check the partition map if it's possible to fit these new footprints in, ignoring any occupied cells that already contain a member of this hierarchy. If it is possible to move the footprints the partition map will be updated accordingly and the requested footprint cells, now free, are returned to Place Object as available space to place the object it is currently working on. If the parent is, in turn, a leaf of another object then that object will be asked to move accordingly and so on.

Find Space will, given a list of potential coordinates in the partition map and the size of the footprint that needs to be fit in, attempt to find a coherent rectangular space that matches the size of the requested footprint. Scanning is done using a by-row, by-col set of nested loops. If space is found it will return the partition map coordinates of the footprint for Place Object to place the object into. The behavior of this function can be altered depending on the object Space Reservation Mode setting. Under normal circumstances space will be required to be inside the room, and unoccupied by anything else. However, by setting the mode to No Reservation Find Space will only make sure that the object is inside the room, and can be fitted while ignoring any overlapping objects. This can be useful for carpets and other low profile objects, for example. At present objects are represented as a rectangular footprint which causes some unexpected behavior with objects that are better represented with a T- or L-shaped footprint.

Populate Leaves goes through all the leaf rules for the placed object. Each object can be defined as Required or Optional and have a percentile score which determines how likely this leaf is to appear, and are required to specify which side of the parent object this leaf will appear; Left, Right, Front, Back, Above or Below. Objects requesting to be spawned Above or Below do not reserve any space in the partition map at this stage and are expected to be used for ceiling lamps and carpets respectively–objects on a table are instead populated using a child PDV as explained later. Because each leaf rule contains a link back to an object database and row of the same kind normal object spawning uses these leaf rules are passed to a new sequence of Place Object with the newly spawned object as the parent, thus overriding the normal placement rules for these objects with that of the leaf rules. If a leaf object fails to spawn, that is if Place Object returns false, it may cause its parent to fail evaluation if the leaf is a required object. When this happens the parent will delete itself

and all already placed leaves, and notify its owning Place Object that it has failed to spawn. If the leaf succeeds in spawning the Place Object responsible for that leaf will attempt to spawn leaves in turn, stepping down a hierarchical object level. Once all leaves are satisfied it will continue by requesting that any child PDVs it governs initialize and start.

Any **Child PDVs** that are held within the newly spawned object are told to initialize and then start laying out on a separate thread, with each PDV running as its own process. This effectively steps down a hierarchical PDV level while the current level returns to Select Object to continue populating itself. Child PDVs can be configured to either invalidate their parent if they fail their own evaluation, or ignore it and leave their parent be. If they invalidate their parent the parent will go back to its parent PDV which will add it to a list of objects to re-evaluate at the end of its own evaluation, or re-evaluate the object immediately if it has already finished its run.

4.9 Implementation

We implemented our system using the "Unreal Engine" [13] game engine version 4.24, from here on referred to as UE4. This allowed us to create a rich and game-like environment to test the implementation under the circumstances that we felt were applicable to a real-time decoration system for games. To allow for easy iterations of the system we built the majority of it using the visual coding system Blueprint (Figure 4) present in UE4 providing quick and easy modifications and logic path analysis without having to recompile code every change and to harness already existing systems for user interaction to save time.

Our implementation in UE4 makes use of the Gameplay Tags feature which allows hierarchical tags to be applied to anything and can be seen as a middle-ground between simple strings and enumerations of values. They allow us to designate an environment as "PDV.Environment.Kitchen" and an object type as "PDV.Object.Furniture" with the possibility to add sub-tags such as "PDV.Object.Furniture.Sofa" to further specify things. Additionally they help with defining various helpers such as "PDV.Direction.Forward" when creating placement semantics.

We built this implementation around the premise of populating rooms. This means that all root objects in the world are rooms and each room is defined to have a PDV covering its interior and a room type to determine the environment type. Because of this the game will locate all room objects when the game starts and figure out which room the player is inside of. This room will then be asked to start decorating and follow the flow of the algorithm outlined in 4.8; it will gather the objects which should appear in this particular type of room and then begin populating the space accordingly.

All object data is contained in DataTable type assets. Each row in a PDV DataTable contains the Name of the object, an array of possible Actor Classes that represent the object to allow variation of the same kind of object with different models and colors, placement semantics such as if the object should be against a wall, floating or a leaf and a map of possible leaves that it itself can spawn. This map uses a cardinal direction as a key, allowing only one object to be spawned in either direction, and a structure for each key which contains leaf placement semantics and a reference to another row in the same or a different DataTable to denote the leaf object itself. Finally the row contains a map of environment scores; a key-value dictionary with a Gameplay Tag denoting the environment type, and a float value denoting the percentile chance the object will appear in that environment.

The PDV itself is represented by a custom box volume actor, an outline shape with configurable extents, which can be added to any object either as a component or directly to the world. It has additional properties which allow the user to configure the relevant DataTable for it to select objects from and the type of objects it should contain, and implements all the required functions to decorate per the algorithm. Our implementation is built in such a way that all PDVs expects there to be a Room actor at the top of the chain which contains a Root PDV. They will read DataTable settings from this PDV unless their own Override Table is set and can as a result act as an extension of the Room itself–allowing the user to configure one type of Environment per room and all PDVs will react accordingly.

UE4 expects actors to face forward in the X axis with their Y axis to the right, and Z up and as such we opted to use this same method of defining the cardinal directions of each object spawned by the

Figure 4: Blueprint Example (Find Space Function) with a section zoomed in.

PDV, as well as using these axes to define the direction of the PDV itself. All creation is done in a spatial environment local to the PDV to allow calculations to be easy to both implement and debug and it is not until the actual placement of the actor that any rotation and movement compensation is done. To further visualize the decoration process inside the room we implemented a visual representation of the partition map and the population process as well as a method to step the decorating process so that we could analyze its behavior. This made it much easier to identify logic errors in the system during development and allowed for a much faster iteration time. Figure 5 shows an example of a a room having been subdivided with visual cells. While a room is being populated the color of the cells are updated to reflect the state which they are in; green represents empty, blue tentative and red occupied. Figure 6 shows the same room in a perspective view to show the in-game representation of the subdivision with furniture.

Figure 5: Orthographic view of a subdivided room with objects.

Figure 6: Example of a room with visual subdivision.

5 Results

The artifact itself proves to be functionally sound. It is capable of generating any number of different rooms provided a seed and a set of objects to place. When given a room of the same size, the same source data, and the same seed it will always create the same room with the same placements except for random rotations applied to add some

variance. These could, of course, also be based on the same seed but we felt that this would make the rooms less interesting. Give the same room size and source data system will produce a unique room for every seed. It is capable of handling rooms of any size but issues may appear if rooms are very small and have requirements for furniture that take up a lot of space. Further the artifact is capable of handling any number of windows and doors in the room but, again, issues may arise if there's a huge amount of doors or windows and the room has rules requiring furniture which must be placed against a wall and is too tall to be placed below a window. These problems do not occur if the room has a normal amount of either. For example an office meeting room with one or two glass walls should probably not have furniture against those walls but can have a conference table in the middle and supporting furniture along the non-glass walls of the room.

This being said the artifact is highly dependent on several factors that affect complexity. These include the size of the space, the size of the objects that are intended to populate it, the complexity of the placement rules of these objects and the hierarchical connections between them. More complicated hierarchical rules can cause the space to be reevaluated multiple times in an attempt to find a working solution. A leaf at the bottom of the chain may lead to a whole leg of the hierarchy being invalidated and requiring reevaluation. This can also happen multiple times which leads to time consuming processes and possible hitching of the game during live decoration. However, after an environment has been populated it can be reduced to a small structural representation of position and object class which lets the environment be quickly rebuilt on subsequent visits while also remembering changes that the visitor may have made, such as moving furniture, adding or removing objects, and so on. This does cost memory but no where near the amount required if objects remained in a loaded state when the player is far away from the environment, and if the content list is streamed with the rest of the level the added costs would be negligible. For a very densely decorated space it may be prudent to generate them at design time and store the structural representation before shipping the product.

Additional time is also required to build intelligent placement rules when assets are created compared to not using the PDV system, and the environment designer may also need to look over the environment to make sure that nothing is placed in strange ways or locations. The time spent by the environment artist assembling placement rules and then conducting quality assurance on a generated environment is negligible compared to time spent assembling the environment manually and more so when comparing with the time spent configuring materials, colliders and so on.

A consequence from using bounding boxes to approximate furniture leads to issues with objects that may be represented as the union of two rectangles or even more complex shapes, for example corner desks and sofas. These objects would appear to take up a large amount of space inside the decoration system when they are approximated by a bounding box despite having ample space where things could be placed. This can be worked around by extending the implementation to account for these kinds of objects; flagging the empty spaces as Tentative during placement of the object or not transferring them to the main cell array after the object is placed, only using the actual footprint. This would, of course, add some extra difficulties in moving furniture around if any child objects require extra space. Another option is to reserve this tentative space for leaf objects so that they can be allowed to exist inside the bounding box of the parent but this, again, requires the objects to be a certain shape so that the resulting tentative space will be large enough to accommodate a leaf. Likewise the use of box volumes to decorate spaces requires the space to be able to be represented as such. One can work around this by having more volumes per space or by modifying the algorithm to account for rooms with irregular polygonal floor plans. One method for scanning for space for irregularly shaped objects could be to place it according to its rules and then sliding it until space is either found or the object goes outside of the volume.

Another limitation is found in the strength of the system. It is possible to define overly complex hierarchies which either lead to very narrow rules or extremely long chains of objects which need to be spawned. This can cause slowdowns or result in partially empty, or even completely empty rooms as a result. Care must be taken when formulating both placement and hierarchical dependencies and it may often pay off to look at real world scenarios

if this continues to be a problem.

We could not reach the expected level of granularity in the decoration process during the time we had for this project. A lot of trial and error has led us to a point where the decorator creates pleasing enough environments which require minor editing after they have been completed. It may have placed a piece of furniture in an awkward location or offset something that would look better centered. In addition to this it can sometimes take very long to completely solve a room depending on the composition of furniture that the randomizer selects. While one could argue that this fits within the parameters of the environment artist performing touch-ups on the environment after it's been procedurally generated we feel that it is possible to reach a better level of quality with more time. This could be spent both fine-tuning the object relations and the implementation of Find Space.

Regardless of the points made above we still think that we've constructed a useful method of decorating a space in a hierarchical manner that can be used in video games. The artifact is still evolving and we have several ideas for improvements to both the selection and placement parts of the algorithm such as handling non-rectangular pieces of furniture and environments and better coherency checking. We feel we have succeeded in what we set out to do even if we did not reach the level of functionality we would like to have.

6 Analysis

To help us get a broader view of our own results we conducted a blind test evaluation of the content that our system can produce. One of our goals with this method of procedurally populating rooms with content was to make it appear believable from the point of view of the player and thus make it appear as if the environment had been built by hand. The blind test was designed to gather information from participants without directly informing them of the goal of the evaluation. Our reasoning behind this method was that procedural environments in video games often look very chaotic or very uniform such as a room full of stacked boxes, identical hallways or carbon-copied offices with the same layout. With the information gathered from this test we hoped to establish if, despite the present shortcomings of the system, it was capable of generating the environments that we had aimed for. Because performance on real-time decoration of an environment cannot be evaluated by images we conducted a separate analysis of this.

Though our system was initially built with a stepping mechanic to allow us to see progress of the room being generated the removal of this mechanic presented some interesting results. The system is capable of filling a reasonably sized room with content in less than a second when run in the editor, that is without "cooking" the blueprint code to C++ first. This was quicker than expected with a scripting language and was a pleasant result. While testing with several rooms in a row and asking the system to populate them as the player ran along a connected corridor we encountered occasional hitching of a couple of milliseconds if the player travelled fast enough. We believe this could be alleviated by compiling the blueprint code or porting the PDV to C++. The hitching was not present when a room was deleted and then reloaded after being far enough away from the player for a set amount of time.

6.1 Evaluation

To evaluate the generated environments we created four different interior rooms. Three rooms were generated by the system (Figure 7, 8, 10) and one was manually built from scratch (Figure 9). Most of the generated scenes were manually edited to remove one or two errant objects within the limit of what we considered to be the level of touching up an environment designer would have to do. The focus remained on these environments being created with as little post-generation input as possible. Rooms were created with different seeds to ensure that they had different layouts but kept the same object source. Image 3 was manually created and the other 3 images generated by the system. The floor plan of the manually generated room was intentionally made slightly different to give an artificial difference in the rooms other than the contents. This was intended to see which feature of the rooms drew more attention; the size of the room or the appearance of artificial decoration.

24 participants took part in the study. They were first asked to look at these four images and then rank them in order of preference from 1 to 4 where

1 was their most preferred image and 4 was their least preferred. Next they were asked to explain what had made them select the image they had ranked as number 1, and finally how many hours they spent playing video games per week. The overwhelming result was that participants preferred image number 3 over the others, which may indicate a heavy preference towards the manually composed room.However, when asked to provide constructive feedback as to why they preferred this image the results provided yielded some interesting data points.

Figure 7: Image 1 in the evaluation, generated

Figure 8: Image 2 in the evaluation, generated

Participants commented that larger rooms are better or that certain furniture features were preferred over others, stating comments such as "Bigger couch" or "Room feels the most open". Comments remained revolving around the placement of furniture or the grouping thereof. On Image 1 a participant commented that they "Liked the spacing and flow" of the image, and that "everything is in order". Even the negative comments on the manually constructed environment revolved around the selection of furniture itself such as disliking the corner computer desk style. Over all one user stated that they felt it was "good Feng Shui". No comments suggested that anyone felt that the rooms were generated by a computer and no user raised this question during their testing. After the test was concluded and this fact was revealed a number of participants stated that they "were baffled that this was the case" and who thought we were looking for optimal furniture placement.

Figure 9: Image 3 in the evaluation, manual

Figure 10: Image 4 in the evaluation, generated

We feel that these responses provide a passing grade for the system; it is capable of decorating spaces in a manner similar to that of a level designer and that the focus laid on the size of the room rather than the contents. This suggests that nothing appeared to be out of the ordinary for the participants and that the rooms did not have a particular tell which drew the attention to it being created automatically.

The amount of time spent playing video games also had an interesting correlation with room preference. Figure 12 shows a cross-comparison between playing frequency and which image they preferred. Here the generated images 1,2 and 4 com-

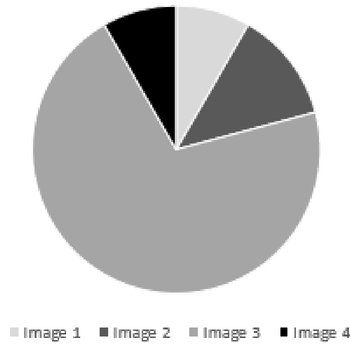

Figure 11: Participant Image preference.

bined vs the manual image 3. It is interesting that this might suggest that "gamers" might be a bit more forgiving than people that play less video games. As they could be suspected to be the main consumers of video games, both with generated interiors and without, this could interesting to investigate further. Video gamers being more susceptible to procedurally generated content and the possible flaws there in is rather interesting to us and may be a field to peruse in further studies; being able to find where the line is actually drawn for visually generated content before it becomes disliked.

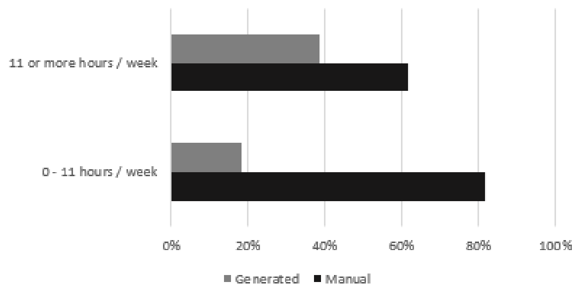

Figure 12: Comparison of playing frequency and preference of generated vs manual decorated room interior.

7 Conclusion

We can conclude that the Procedural Hierarchical Decoration system that we present is capable of generating room interiors and decorate these rooms. The resulting room interiors seem plausible and the results from our evaluation suggests that there's no direct tell that these rooms have been procedurally populated. However, the current version of the system is not without errors at this moment and still puts objects in awkward positions now and then. The system could benefit from more work on the placement algorithms and from enhanced features to handle non-rectangular spaces, although it is still suitable in the context of dynamic generation of room interiors both at design- and run-time. We have already started to translate the blueprints to C++ to be able to use more efficient methods for the calculations and to leverage the advantages that the language offers in an effort to see if the limitations of the Blueprint language has any effect on its behavior.

We feel that the Procedural Hierarchical Decoration system is a small but interesting step towards the goal of being able to fully rely on an automated system that generates plausible interiors of buildings, at run time, in the proximity of the player.

8 Future Work

As future work, we would like to couple the system with one that generates procedural floor plans that the PDV can then decorate. This would give a more complete module and expand the usability of the Procedural Hierarchical Decoration system. It would also be interesting to further improve the system so that the preference for a procedurally decorated space is on par with a manual space, but further testing would be required to establish if a participant feels that the space truly is generated or not. Our current data suggests that they cannot tell the difference but since this is only one single test it is not enough data to make any definitive conclusions. In hindsight the difference in room sizes may have affected the results negatively, and in favor of the manually created room, and while another test was planned to ascertain if this was the case we did not have the time to do so. Another thought would be to perhaps try using

a semantic approach, i.e have semantic values associated to furniture, in order to place and arrange these in a more meaningful and human-like way. This might be interesting to investigate, although it would probably require a major restructuring of the system. It would also be interesting to compare this system to other procedural decoration methods to investigate which issues they have which are solved by this system, and vice versa.

The user study taught us that the system already generates believable spaces that require minimal touch ups to be ready for use. However, it also shows that we have some ways to go when assembling placement semantics and that we also may have to look over how these semantics are interpreted by the system. It may also be prudent to, in addition to this, create a system that makes sure that a logical path through the room is available. This would help add to the realism of the space and would, presumably, help the passability of the environment. One could ensure that there's always a set amount of unoccupied cells between those that have been claimed by a non-related object such as a sibling or leaf of a sibling so that the player can traverse the area. We believe that the preference shown for larger spaces during the study ties into the passability of the room. Future studies could include a passability score of the room and should also be made with rooms that are of the same size.

References

[1] T. Tutenel, R. M. Smelik, R. Lopes, K. J. D. Kraker, and R. Bidarra, "Generating consistent buildings: a semantic approach for integrating procedural techniques," *IEEE Transactions on Computational Intelligence and AI in Games*, pp. 274–288, 2011.

[2] A. Dahl and L. Rinde, "Procedural generation of indoor environments," p. 53, 01 2008.

[3] R. Lopes, T. Tutenel, R. M. Smelik, K. J. D. Kraker, and R. Bidarra, "A constrained growth method for procedural floor plan generation," in *in Proc. 11th Int. Conf. Intell. Games Simul*, pp. 13–20, 2010.

[4] B. Bradley, "Towards the procedural generation of urban building interiors," 2005.

[5] T. Tutenel, R. Bidarra, R. Smelik, and K. J. de Kraker, "Rule-based layout solving and its application to procedural interior generation," *Proceedings of the CASA Workshop on 3D Advanced Media in Gaming and Simulation (3AMIGAS)*, 01 2009.

[6] id Software, "Doom," 12 1993.

[7] Cloud Imperium Games, "StarCitizen." `https://cloudimperiumgames.com/`.

[8] P. Wonka, M. Wimmer, F. Sillion, and W. Ribarsky, "Instant architecture," *ACM Transaction on Graphics*, vol. 22, 07 2003.

[9] P. Müller, P. Wonka, S. Haegler, A. Ulmer, and L. Van Gool, "Procedural modeling of buildings," *ACM Trans. Graph.*, vol. 25, p. 614–623, July 2006.

[10] E. Hahn, P. Bose, and A. Whitehead, "Persistent realtime building interior generation," in *Proceedings of the 2006 ACM SIGGRAPH Symposium on Videogames*, Sandbox '06, (New York, NY, USA), p. 179–186, Association for Computing Machinery, 2006.

[11] T. Germer and M. Schwarz, "Procedural arrangement of furniture for real-time walkthroughs," *Computer Graphics Forum*, vol. 28, no. 8, pp. 2068–2078, 2009.

[12] K. Peffers, T. Tuunanen, C. Gengler, M. Rossi, W. Hui, V. Virtanen, and J. Bragge, "The design science research process: A model for producing and presenting information systems research," *Proceedings of First International Conference on Design Science Research in Information Systems and Technology DESRIST*, 02 2006.

[13] Epic Games, "Unreal Engine." `https://www.unrealengine.com`.