Extensible Multi-Domain Generation of Virtual Worlds using Blackboards

Gaetan Deglorie[1], Rian Goossens[2], Sofie Van Hoecke[1] and Peter Lambert[1]
[1]*ELIS Department, IDLab, Ghent University-iMinds, Sint-Pietersnieuwstraat 41, B-9000, Ghent, Belgium*
[2]*ELIS Department, Ghent University, Sint-Pietersnieuwstraat 41, B-9000, Ghent, Belgium*
{*gaetan.deglorie, rian.goossens, sofie.vanhoecke, peter.lambert*}*@ugent.be*

Keywords: Procedural Modeling, Blackboard Architecture, Heterogeneous Modeling

Abstract: Procedural generation of large virtual worlds remains a challenge, because current procedural methods mainly focus on generating assets for a single content domain, such as height maps, trees or buildings. Furthermore current approaches for multi-domain content generation, i.e. generating complete virtual environments, are often too ad-hoc to allow for varying design constraints from creatives industries such as the development of video games. In this paper, we propose a multi-domain procedural generation method that uses modularized, single-domain generation methods that interact on the data level while operating independently. Our method uses a blackboard architecture specialized to fit the needs of procedural content generation. We show that our approach is extensible to a wide range of use cases of virtual world generation and that manual or procedural editing of the generated content of one generator is automatically communicated to the other generators, which ensures a consistent and coherent virtual world. Furthermore, the blackboard approach automatically reasons about the generation process which allows 52% to 98% of the activations, i.e. executions of the single-domain content generators, to be discarded without compromising the generated content, resulting in better performing large world generation.

1 INTRODUCTION

As consumer expectations put increasing pressure on the video game industry to improve the visual quality of games, game content production today focuses on increasing the visual detail and quantity of game assets used in virtual worlds. The creation of game content, including but not limited to 2D art, 3D geometry (trees, terrain, etc.) and sounds, largely remains a manual process by human artists. Using more artists to produce a larger but still coherent virtual world becomes increasingly infeasible, not to mention the associated increase in production costs. Procedural content generation (PCG) is the generation of (game) assets through the use of algorithms (Togelius et al., 2013a). This allows for a large increase in the size and diversity of produced content without an associated increase in cost.

However, previous work on PCG focuses primarily on single-domain content generation, i.e. algorithms that only generate one specific type of asset. Multi-domain content generation integrates these methods but producing coherent content at a larger scale remains challenging, and focuses mostly on ad-hoc solutions for specific use cases (Smelik et al., 2011; Dormans, 2010; Kelly and McCabe, 2007). Although these methods generate varying types of content in an integrated manner, the generation process targets a specific mixture of content and generates it in a specific sequence and manner. This makes these solutions less suitable for creativity industries, such as game development, because each project comes with highly varying design constraints and targeted content domains. Although the motivation for this work stems from the game development domain, our work is more broadly applicable to other domains such as animation, movies, simulation, etc.

In this paper, we introduce a blackboard architecture as a solution to this problem that allows PCG methods to cooperate while manipulating highly heterogeneous data to create a coherent virtual world. Our approach is extensible with any PCG method and allows edits of the generated content to be automatically communicated to other content and procedural generators. Furthermore, our approach supports artists and designers by allowing the reuse of integrated PCG methods across different use cases.

The remainder of this paper is as follows. In Sec-

tion 2 we give an overview of previous approaches for multi-domain content generation and the concepts and extensions for blackboard systems. In Section 3 we elaborate upon our architecture. Then we show several design considerations when using our approach to generate virtual worlds in Section 4. In Section 5 we evaluate the extensibility of our approach, as well as the editability of the generated content and the generation performance. Finally, we present the conclusions and future work in Section 6.

2 RELATED WORK

We introduce a multi-domain content generation approach based on blackboard systems. The discussion of single-domain procedural methods lies beyond the scope of this work, for an overview of procedural methods we refer to a recent survey book (Shaker et al., 2016).

2.1 Multi-Domain Content Generation

A prominent approach to multi-domain content generation is the waterfall model used in urban contexts (Parish and Müller, 2001; Kelly and McCabe, 2007) and game level generation (Dormans, 2010; Hartsook et al., 2011). Both the work by Parish and Kelly integrate several procedural methods (road network generation, plot subdivision and building generation) sequentially to create a complete and coherent virtual city scene. Dormans combines the generative grammars to generate both mission and space of game levels, where the mission is generated first and the space is mapped onto it. However, the rules used in the system are game-specific and as such cannot be easily reused. In a similar approach, Hartsook uses procedurally generated or manually authored stories or narratives to generate the spatial layout of a game level. Waterfall model approaches, however, assume a predefined order between generators. This imposes constraints from each layer to the next, which in turn reduces the re-usability of the system. The sequential and tightly coupled nature of such systems makes them hard to extend to new use cases with new generators. Furthermore, editing content at a certain level requires that content at lower levels should be entirely regenerated, although creating (non-extensible) ad-hoc editing operations is still possible (e.g. the work by Kelly).

Alternative approaches to multi-domain content generation include data flow systems (Silva et al., 2015; Ganster and Klein, 2007), combining procedural models into a meta-procedural model (Gurin et al., 2016a; Gurin et al., 2016b; Grosbellet et al., 2015; Genevaux et al., 2015), declarative modelling (Smelik et al., 2010; Smelik et al., 2011), answer set programming (Smith and Mateas, 2011) and evolutionary algorithms (Togelius et al., 2013b). Silva augments generative grammars by representing them as a data flow graph. This allows them to add additional content domains such as lights and textures as well as new filtering, grouping and aggregation features. Although the data flow approach splits the procedural generation process into separate nodes, an additional requirement for data flow graphs is that every node needs to know in advance how it will interact with other nodes. This adds overhead to creating nodes as the designer potentially needs to revisit previous nodes. Furthermore, all editing operations need to be translated from the content domain into the procedural graph domain which requires additional expertise from the user. The work by Guerin, Grosbellet and Genevaux on meta-procedural models focuses on two separate directions. Firstly, meta-procedural modeling of geometric decoration details involves generating details such as leaves, pebbles and grass tufts for pre-authored environments. Secondly, meta-procedural modeling of terrains involves generating terrain height maps, waterways, lakes and roads in an integrated manner. This is achieved by using a common geometric representation set, i.e. elevation functions, which allows these different terrain features to be integrated. Both methods impose a specific ordering or structure on the generated content, which means that they are incapable of handling different domains. Smelik combines the generation of different aspects of a virtual world, including road networks, vegetation and terrain. Instead of a waterfall model, all generation occurs independently and is subsequently combined using a conflict resolution system. Their editor provides an intuitive way of editing for their specific content. However, adding new generators to their system is difficult. Smith defines the design space of the procedural generation problem as an Answer Set Program (ASP). By formally declaring the design space, they can define new procedural methods for a variety of content domains. Togelius formalizes the entire game level into a single data model and use multi-objective evolution to generate balanced strategy game levels. Both the evolutionary and ASP approach additionally require a formalization of the underlying data model. Changing the underlying constraints or adding new procedural methods also means updating the data model manually. Additionally edits to the generated content cannot be communicated back to such systems.

2.2 Blackboard Systems

The blackboard system is a technique from the domain of Artificial Intelligence (AI) used to model and solve ill-defined and complex problems (Corkill, 1991). Analogous to a real blackboard on which several researchers work on a problem, the system uses knowledge sources specialized in different tasks. A control system ensures that the knowledge sources are triggered at the right time to avoid conflicts.

Blackboard systems are a good solution for design problems as they can handle problems that have to work on heterogeneous data. Blackboards are therefore used in semantic problem solving (Verborgh et al., 2012), game design (Treanor et al., 2012; Mateas and Stern, 2002) and poetry generation (Misztal-Radecka and Indurkhya, 2016). Previous work does not include the use of blackboards for procedural generation of virtual worlds.

3 BLACKBOARD SYSTEM FOR PROCEDURAL GENERATION

Current multi-domain content generation approaches are hard to extend, impose a predefined order of generation procedures implicitly constraining the whole generation process or regenerate large parts when editing content at a certain level. To solve these problems, we introduce a blackboard architecture that allows PCG methods to cooperate while manipulating highly heterogeneous data to create coherent virtual worlds.

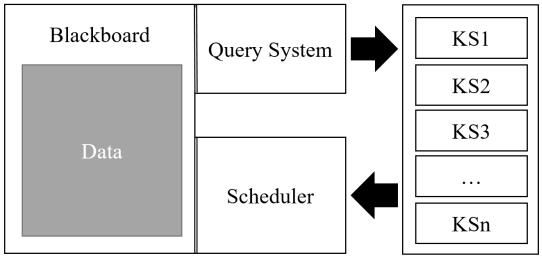

Figure 1: Overview of the blackboard system.

Figure 1 shows an overview of the proposed method. Our system consists of four main components: the blackboard, a query system, PCG knowledge sources (KS) and a scheduler. The PCG knowledge sources create new content based on existing content or data on the blackboard. Using the query system they can access this data and additional contextual information if needed. The execution of each PCG knowledge source is triggered by the occurrence of the necessary input data, i.e. the input type. Data generation is not immediately executed however. Instead the execution of a PCG knowledge source is temporarily buffered in an event called a knowledge source activation. All knowledge source activations are collected by the scheduler. The scheduler can then process the order of execution of the events and filter out unnecessary activations where needed. The remaining activations are executed and change the state of the data on the blackboard. In the following subsections, we will discuss each component in greater detail.

3.1 Blackboard and Data Model

The blackboard stores all the data, i.e. all generated content instances and additional input information. In contrast to previous work using data models where the designer needs to manually create and manage them, we instead opt to automatically generate our model based on the input and output data types of the knowledge sources. The model is automatically created, before the content generation process is started, using the selected knowledge sources that were chosen to generate the virtual world. This model is used to structure the generated data and to help inform the scheduler (which will be discussed in Section 3.4). Figure 2 provides an example of our data model as it was built for the evaluation.

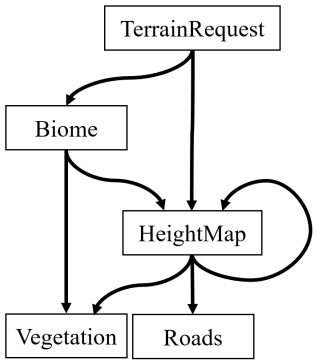

Figure 2: Overview of the blackboard data model.

As each knowledge source will generate new content based on previous content from the blackboard, this implies a dependency from one piece of content to another. These dependencies are encoded as the edges of a directed graph used to store the generated data. For example, the placement of trees might depend on the placement of the road network, to avoid placing a tree on a road instead of next to it.

Our data model allows cyclical dependencies in the resulting content. This increases the expressiveness of the overall system as content generated further in the generation process can affect and improve

earlier generated content (Togelius et al., 2013a). Although we support cyclical dependencies at all levels of our system, our use cases do not extensively test the robustness of including cycles, as cycles can introduce potential deadlocks or infinite knowledge source activations that create an unstable virtual world. We suggest careful design of the virtual world generator, taking care to avoid these unstable situations. A detailed exploration of cyclic dependencies will be addressed in future work.

3.2 Query System

The query system is used to access and retrieve information from the blackboard. It supports direct queries (e.g. retrieve terrain height map) or contextual queries (e.g. retrieve terrain only from Arctic biomes). The query system is currently implemented as a basic database with basic select and where clauses. Queries are executed based on data type, this means that a formal model is needed for all PCG knowledge sources to interact in a coherent manner.

The contextual queries are performed by searching the directed graph for ancestors and children based on the required information. Additionally, a contextual query might encode a requirement for the parameters of a piece of data instead of its relation to other nodes inside the graph. For example, a terrain data request could contain the added context of only requiring data of a minimal elevation.

3.3 PCG Knowledge Sources

PCG knowledge sources encapsulate procedural methods, preferably single-domain algorithms such as L-systems or coherent noise generators as these improve re-usability. Additionally, we distinguish three types of PCG behaviour: (1) addition of new content instances, (2) modification of existing content instances and (3) deletion of existing content instances. A PCG knowledge source is required to implement one or more of these behaviours to be supported by the blackboard system. As stated earlier, a knowledge source does not immediately execute when it retrieves data from the scheduler, instead it produces a knowledge source activation. Knowledge source activations store their operation (i.e. addition, modification or deletion) in an event and send it to the scheduler for execution. The separation into three types allows us to reason about what the generators plan to do and optimize the order or even remove some unnecessary activations (see next section). However, this requires the PCG knowledge sources to be stateless, i.e. they should not remember what content instances were previously generated by them, as manipulating the order or occurrence of activations would create a mismatch between the internal state of each knowledge source and the state of the blackboard.

3.4 Scheduler

The scheduler controls the execution order of PCG knowledge sources and handles the execution of knowledge source activations. The execution order is determined by automatically calculating a priority for each knowledge source in the system. This priority is derived from the blackboard data model. The priority order can be determined by performing a topological sort of the directed graph. However, topological sorts only work on directed acyclic graphs (DAGs). By condensing cycles in the graph into single vertices, we can make any directed graph into a DAG.

Based on the determined priorities, the knowledge source activations are collected in a priority queue. This ensures that when a knowledge source activates, all work resulting in its input type will be finished. The next step is merging and removing unnecessary knowledge source activations, done in accordance with two rules: (1) deletion cancels additions and modifications and (2) modification can be merged with an addition event to form a new altered addition. Merging activations reduces the amount of activations to be executed, without changing the results, thus increasing the generation performance of the system. This will be evaluated in Section 5.4.

4 DESIGN CONSIDERATIONS

In this section, we will discuss the insights obtained from the implementation of our use cases. As each knowledge source encapsulates a procedural method and only communicates through the data on the blackboard, it provided ample opportunity to create reusable modularized behaviours. The scope of our work is virtual world generation. For this, we propose a set of reusable abstractions, that we dubbed knowledge source design patterns. First we will discuss what considerations can be made in terms of compartmentalizing generation processing to improve modularity when designing knowledge sources. Next, we will provide an overview of the knowledge source design patterns that were useful when designing use cases for virtual world generation.

4.1 Modularity of PCG Knowledge Sources

Each knowledge source encapsulates a procedural method capable of generating a specific piece of content. One could encapsulate any stateless state-of-the-art procedural method into one of these modules, e.g. putting an entire L-system and turtle interpreter into a single module. However, naively encapsulating algorithms will negatively impact performance. The PCG knowledge sources are only dependent on their input and output data types. For example, a knowledge source might produce one or more elements of type C from an input of type A. This knowledge source can be replaced by two (or more) knowledge sources, e.g. one knowledge source that produces B from A and another that produces C from B. As long as the intermediate data type differs from the starting input and output types no conflicts will arise within the generation process.

From a design perspective, we can modularize knowledge sources in three different ways: (1) by output type, (2) by input type and (3) by generation process.

1. The output type can be made more generic. By splitting a knowledge source in two, an intermediate data type can be created which contains more generic information which can be reused for subsequent processes. Figure 3 shows an example of splitting a forest generator into an object distribution generator and a tree generator. We can then reuse the locations produced by the object distribution generator to place other objects.

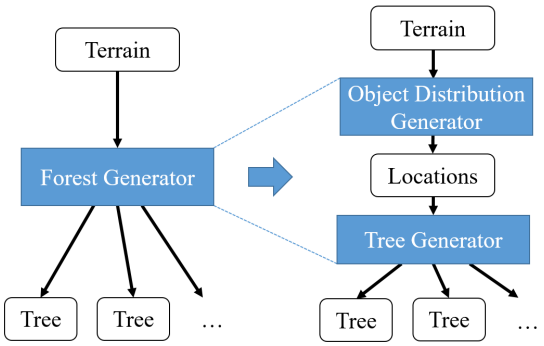

Figure 3: Example of modularization by output type.

2. The input type can also be made more generic. Instead of making specialized processes for each input type, it is more beneficial to split these into a specialized converter to an intermediary type and create a more generic process for this type. Figure 4 shows an example of generating bird nests in trees for both fir and jungle type forests. We can make a specialized parser that converts this tree information into a more general tree model, such as a branch list. From this list a generic bird nest placer can be used.

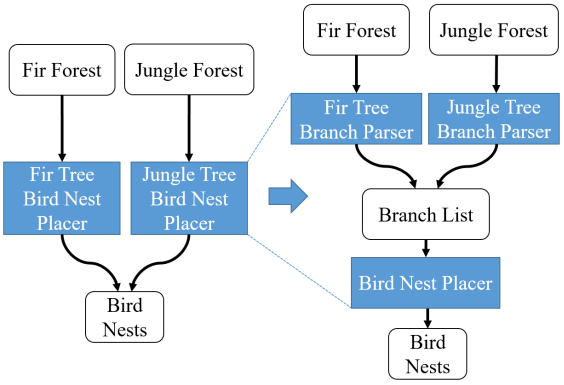

Figure 4: Example of modularization by input type.

3. Given the same input and output data types, multiple knowledge sources can be created that contain different algorithms. For example, a knowledge source calculating the minimal spanning tree using Kruskals algorithm, which is faster for sparse graphs, could be switched out with another one using Prims algorithm which is faster for denser and larger graphs.

These three types of modularity should ideally be used together, maximizing the reuse of modules across different domains. As will be discussed in Section 5.4, this will impact the performance of the entire system.

4.2 Knowledge Source Design Patterns

The different design patterns obtained from the implementation of our example scenes for evaluation (see Section 5.2) are:

- The **converter** simply converts data types from one type to another. These are typically used to convert specific types into generic types or vice-versa (e.g. generic bird nest placer).
- The **distributor** generates a specific amount of instances from an input type, according to a distribution function. These are typically used for object placement in a certain search space.
- The **modifier** simply modifies data of a certain type, i.e. manipulating its parameters. For example, modifying the height of a mountain at a certain location.
- The **constraint** ensures that data has been produced in the right environment by allowing it to

modify or delete parent data if it does not adhere to this constraint.

- The **collector** collects instances of a specified data type and puts them into a collection. This is useful when you want to perform an operation on several instances of the same data type.

Further identification of these design patterns will support the applicability of our approach in future work.

5 EVALUATION

The evaluation of procedural generation methods remains a challenge. Content representation has not been standardized, i.e. a geometric object can be represented in a variety of a ways (e.g. voxel representation, triangle meshes or billboard images). This means that comparing the resulting content is often approximate at best. Furthermore, different procedural methods can focus on specific features, e.g. generation performance, content quality, extensibility, etc.

We chose to compare our approach to Esri's CityEngine, a state-of-the-art urban city generator, to give the reader a general idea of what is currently used in industry. CityEngine is also a good example of a waterfall approach to multi-domain content generation; it is a well established commercial tool and has an advantage over our approach in terms of design features, interface accessibility and performance. However, comparing our proof-of-concept in terms of design features or user interface is out of the scope of our research. Instead, we aim to show that the blackboard-based approach is highly extensible and is more robust when editing generated content.

5.1 Test Cases

We created a number of test cases to evaluate the extensibility and the generation performance of our system. These were created using custom modules for procedural content generation of virtual worlds: (1) a generic object placement module allows for objects to be distributed randomly and allows meshes to be placed in the scene; (2) a terrain system allows for complex terrains with customizable and fully independent biomes; (3) a vegetation module leverages L-systems to produce vegetation meshes; and (4) a road module allows for road networks to be built by various sources and supports multiple road strategies, similar to the technique used by Kelly et al. (Kelly and McCabe, 2007). Covering all possible content generation algorithms to create virtual worlds would be out of scope for this paper, however these examples form a representative sample of virtual world generation in general.

Example 1: Forest Scene

The forest scene features a generated height map with variable amount of vegetation objects. By default, the forest scene contains an equal amount of both trees and bushes. We can however remove the bushes or add a third object, e.g. plants. Varying the amount of object types will be discussed further in Section 5.4.

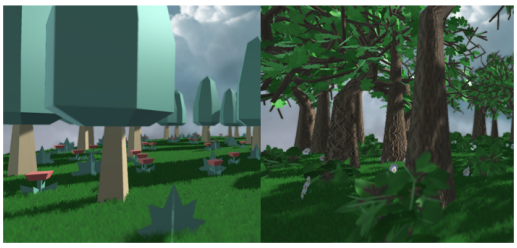

Figure 5: Forest scene using object placement (left) and L-systems (right).

Figure 5 shows an example forest scene featuring 50 trees, 100 bushes and 100 plants. By swapping out knowledge sources the vegetation can be either created by placing predefined objects, or by creating meshes at runtime using an L-system.

Example 2: Road Scene

The road scene generates a height map for a desert environment, with a variable amount of interest point objects. The interest points are combined into a road graph according to two strategies based on previous work (Kelly and McCabe, 2007): (1) straight road, connecting two points with the shortest possible road, and (2) minimum elevation difference, roads with lowest steepness within a maximum allowed extension versus straight roads.

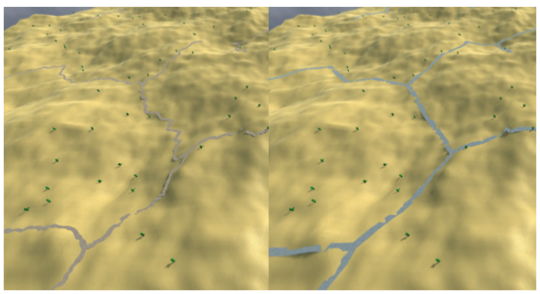

Figure 6: Road scenes using minimum elevation difference (left) and straight strategies (right).

Figure 6 shows an example road scene with the

two different strategies. The different road strategies can be utilized by swapping out knowledge sources.

Example 3: Combined Scene

The combined scene is an example of combining all of the aforementioned techniques. It features a generated height map, different biomes (e.g. arctic, desert and forest), vegetation generation (e.g. using L-systems) and placement and road network generation. This example serves as a "complete" virtual world and a point of comparison with the content produced by CityEngine.

Figure 7: Example combined scene.

Figure 7 shows an example of the combined scene with the placement of vegetation and the choice of road strategy automatically adapting to the biome. It can be noted that our prototype does not feature building generation as is the case with CityEngine. However, CityEngine's tree representation uses pre-authored billboards while we use L-systems to generate full 3D trees with leaves. Thus we argue that these scenes have a similar scene complexity generally speaking.

5.2 Extensibility

Current approaches of multi-domain content generation have limited extensibility. They make choices on what content should be generated, how and in what order. This makes them less suitable for the creative industries, as artists need (nearly) complete control over their design tools. CityEngine, for example, generates the road networks first, followed by plot subdivision in the resulting street blocks and finally generating a building or open area on each plot. This hampers the usability of such a system for different domains or projects. For example, this implies that an artist cannot place one or more buildings first and add a street connecting them to the street network, or create a plot subdivision based on the placement of a set of buildings and roads.

Our blackboard approach however allows a more flexible way of editing the content generation process for *any* combination of content domains, e.g. extending the system with new generators or changing the content dependencies, because our blackboard data model is automatically generated based on the selected knowledge sources. Furthermore, the scene generation does not happen in concrete steps or in a set order. Instead the scene is built in small increments and previously generated content can still be affected by content that has yet to be generated. This allows for more innovation than would normally be possible in a system where rules cannot be dependent on aspects that appear later in the generation order (Togelius et al., 2013a).

Our architecture makes knowledge sources, and consequently the procedural methods, independent of each other where communication is handled implicitly through the blackboard, or more specifically through the data.

5.3 Editing Generated Content

Another key advantage of our approach over waterfall approaches is that it allows newly generated content to edit previously generated content without complete regeneration. In a waterfall-based approach, editing content in a certain layer causes all subsequent layers to regenerate as there is no way to communicate what specifically has changed in that layer and what parts of depending content should change accordingly. This can be partly alleviated by implementing ad-hoc solutions where necessary. For example, in CityEngine, very small translational movements (less than 1 meter) of a street intersection mostly does not regenerate all surrounding building blocks but instead resizes the building plots without having to regenerate the building. However, even slightly increasing this movement causes the subdivision to update, which in turn regenerates the buildings. This means an apartment building on the corner of the street might turn into a small park.

Obviously when changing content in a scene all dependent content should change, however the underlying problem is that content dependencies in a waterfall model are over-generalized. Content naturally depends on each other, but we need to model these dependencies at a more granular level. In our approach, generation is segmented into several knowledge sources which exposes these content dependencies. This way, changes in the scene do not cause complete regeneration, but instead allows a more localized effect where only the properties of the content that should change do change.

Our generation process triggers knowledge sources through changes in data on the blackboard, i.e. additions, modifications and deletions. Thus any modification of data on the blackboard, i.e. the generated virtual world, triggers knowledge sources that depend on that content type and subsequent knowledge sources, cfr. the data model. Figure 8 shows an example of the height map being displaced and the nearby trees automatically updating their vertical position and orientation (to follow the surface normal). In this case, only the vertical position logically depended on the height map, thus the overall horizontal distribution is maintained and all trees are still placed correctly on the surface of the virtual world.

Figure 8: Displacing the height map locally updates the height and orientation of nearby trees.

Important to note here is that this behaviour is observed irrespective of whether the edit was procedural or manual. Because the generation process has been split into different knowledge sources, manual editing of the resulting content automatically causes dependent knowledge sources to update their content. The changes cannot be communicated to the knowledge source that created said piece of content however. Consequently, if another change happens earlier in the dependency graph, the first change will be overwritten.

5.4 Generation Performance

To evaluate the generation performance, we will broadly compare our system with CityEngine. Example 3 from Section 5.2 will be used as a reference to CityEngine's urban worlds. We generate a virtual world of similar size and scene complexity with both tools and compare the execution times. Our framework has been implemented on the Unity Engine (UnityTechnologies, 2016), and all tests were performed on an Intel Core i7-5960X 3.00GHz computer with 32GB of RAM.

Esri's CityEngine (Esri, 2016) creates cityscapes with an approximate virtual size of 12 km^2 about 7500 to 9500 geometric objects in 15 to 30 seconds. Conversely, our approach creates a virtual world of approx. 12 km^2 with 8000 geometric objects in about 4 minutes. Although our system performs 10 times slower than CityEngine, it should be noted that this is for full regeneration in both cases and comparing a commercial optimized solution versus a research prototype. When editing the scene, the changes typically take a couple of milliseconds to at most a couple of seconds for our system, similar to CityEngine.

It should be noted that the resulting performance of our system depends on the number of knowledge source activations. The constant sorting and scheduling of these events introduces an overhead into the system, and furthermore the design choices, i.e. how to split up the generation process into knowledge sources, also impact performance. However, they do not increase the runtime complexity of the underlying algorithms, e.g. a road generation algorithm of complexity O(n^2) remains O(n^2). For example 3, at 8000 geometric objects, we measured on average 6.7 seconds of overhead.

In the next Sections, we discuss the impact of scheduling and modularization on the generation performance of our blackboards architecture.

Impact of Scheduling on Performance

In order to evaluate the impact of scheduling, we generate each example scene (see Section 5.2) with and without event reduction at varying scene complexities. Figure 9 shows the relative number of reduced activations due to event reduction.

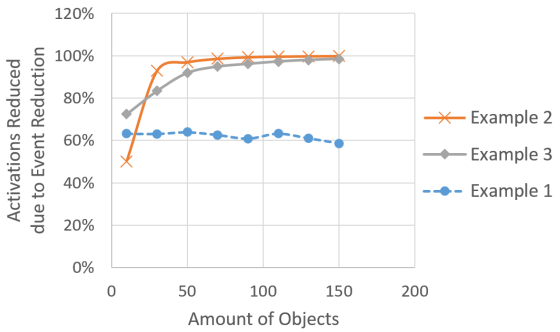

Figure 9: Reduced activation ratio with event reduction for all use cases.

We can see that the activation reduction increases significantly over time for examples 2 and 3, while example 1 remains fairly constant. This can be explained by the presence of the road network generation algorithm in both examples 2 and 3.

Creating a road network in our case involves relatively more knowledge sources than for example forest generation, i.e. 6 steps instead of 3: (1) height map

to point cloud, (2) point cloud to interest point, (3) interest point to point collection, (4) point collection to abstract road graph, (5) abstract road graph to road graph with strategy tags and (6) road graph to textured mesh. Furthermore, the event deletion greatly reduces vertically, i.e. generators using highly sequential or a large amount of dependent steps benefit most from event deletion. In conclusion, the event reduction can reduce the number of activations from 52% to 98%.

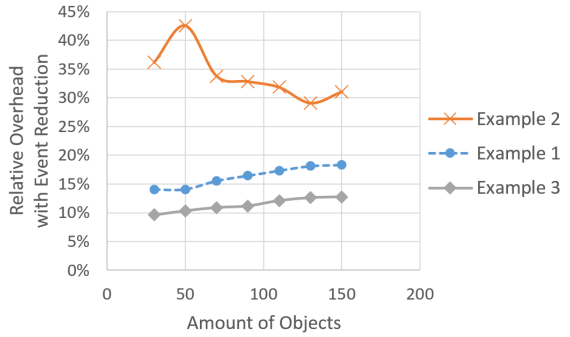

Figure 10: Overhead ratio with event reduction for all use cases.

It is clear that event reduction is beneficial for large scale virtual world generation. However, event reduction also introduces an overhead into the system. Figure 10 shows the relative overhead (i.e. the percentage of time not spent on generation) for all example scenes. We can see that the overhead is between 13% and 42%. Example 2 can be considered highly vertical in terms of content generation, conversely example 1 and 3 are more horizontal, i.e. subsequent knowledge sources mostly reuse content generated by a single knowledge source. For example, the placement of road intersections and tree positions are both derived from a point cloud generated based on the height map. From this, we can infer that the more horizontal, i.e. reusable, the generation process the lower the overhead.

Impact of Modularization on Performance

One can argue however that modularizing PCG generators into several smaller modules will significantly increase the runtime complexity as more modules also means more events. However, although more events are indeed created we will show that the overall runtime can decrease depending on the design.

This test uses example 1, as we stated in Section 5.2 the forest scene can be designed with only trees (single object type), trees and bushes (double object types) or trees, bushes and plants (triple object types). The modularization in all three cases is an example of modularization by output type (see Section 4.1). We will evaluate the performance of example 1 for all three configurations (single, double and triple) with and without the introduced modularization by output type. The configurations are introduced to show the impact of data and knowledge source reuse facilitated by modularization.

First modularization increases the activation count by 200% for the single configuration, by 50% for the double collection and stays about the same for the triple configuration. The highest increase is the single configuration, this is to be expected as we cut up the generation process in more steps without having more knowledge sources that take advantage of it (i.e. reuse). Increasing the amount of object types in our test case however decreases this disadvantage, with an object type count of three almost nullifying the increase in activation count from adding extra modules.

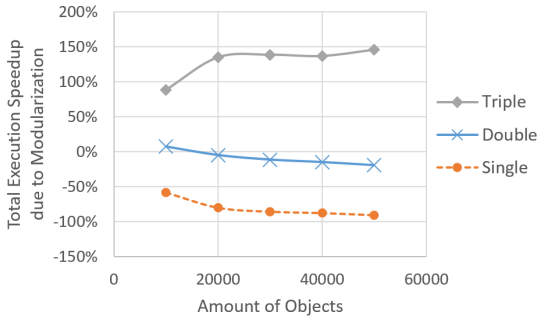

Figure 11: Total execution speed-up due to modularization for different object type amounts.

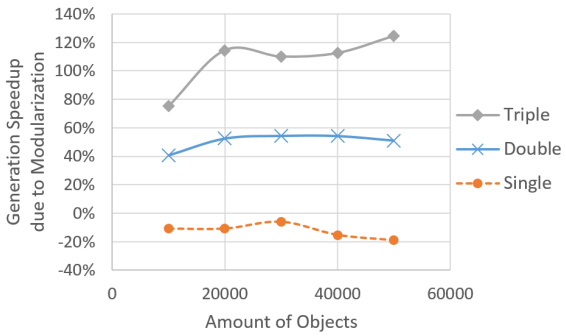

Figure 12: Generation speed-up due to modularization for different object type amounts.

Next we take a look at the execution times, both total (Figure 11) and actual generation (Figure 12). The results show that in the single configuration the generation process is considerably slower, although the generation time does not decrease that much (by about 20% at 50 000 objects) the total execution time at least doubles. However when introducing more object types (i.e. double and triple configurations), the modularized knowledge sources can be reused by

other knowledge sources thus increasing the performance.

Modularization moves work related to converting from one data type to another into a separate knowledge source. The result of this conversion can be used by multiple knowledge sources which would otherwise have to do this conversion themselves. Moving the conversion into its own knowledge source and making the input and output as generic as possible also allows various optimizations which are otherwise not possible. This means that the extra overhead created by modularization due to the extra activations is less than the benefit gained from it.

6 CONCLUSIONS AND FUTURE WORK

To improve the usability of multi-domain content generation, the selected content domains for generation as well as the order and manner in which it is generated should be configurable. We presented an extensible framework for multi-domain content generation of virtual worlds. We have introduced blackboards to the domain of procedural generation to alleviate the current limitations of multi-domain content generation. Encapsulating the different (single content-domain) procedural methods into knowledge sources, allows the system to reason about the generation process which in turn allows optimization of the generation process by eliminating unnecessary generation executions and ordering the remaining based on priority in terms of content dependencies.

We provided an overview of the design considerations when using our method for virtual world generation: modularization and knowledge source design patterns. We have shown the extensibility of our system by implementing a set of 3 different use cases (forest, road and combined environments) which form a representative sample set of virtual world generation. Furthermore, our system facilitates a more stable way of editing generated content, as changes in the data only trigger the specific procedural methods that depend on it. Finally, the generation performance of the system depends on the scheduling system (i.e. event reduction) and modularization design paradigms. Event reduction reduced the number of knowledge source activations by as much as 98% resulting in better performing large world generation. Although modularization increases the number of activations, we proved that the overall runtime can be reduced by intelligent data and knowledge source reuse.

For paths for future research, we suggest five possible extensions. Firstly exploring the behaviour of dependency cycles in the data model. Secondly, improving the editing of generated content by automatically communicating edits to the knowledge source that created said changed content. Thirdly improving the generation performance of the system by for example automatic concurrency of the PCG blackboard architecture. The separation of procedural modelling methods into knowledge sources should provide opportunities for parallelization. Fourthly, data ontologies could be utilized to provide a formalized data format, allowing for more optimizations for scheduling and overall improvement of the coherence of the resulting content. Lastly recursive blackboards could be used for PCG blackboards, where the knowledge sources can contain blackboards themselves. This could be used to enable scoping operations, where certain knowledge sources are scoped within certain regions of the virtual worlds. This would allow finer-grained control over the generation process and allow different types of constraints in different regions.

REFERENCES

Corkill, D. D. (1991). Blackboard systems. *AI expert*, 6(9):40–47.

Dormans, J. (2010). Adventures in level design: generating missions and spaces for action adventure games. In *Proceedings of the 2010 workshop on procedural content generation in games*, page 1. ACM.

Esri (2016). Esri cityengine 3d modeling software for urban environments. http://www.esri.com/software/cityengine/. Consulted August 2016.

Ganster, B. and Klein, R. (2007). An integrated framework for procedural modeling. In *Proceedings of the 23rd Spring Conference on Computer Graphics*, pages 123–130. ACM.

Genevaux, J.-D., Galin, E., Peytavie, A., Guerin, E., Briquet, C., Grosbellet, F., and Benes, B. (2015). Terrain modelling from feature primitives. *Computer Graphics Forum*, 34(6):198–210.

Grosbellet, F., Peytavie, A., Guérin, E., Galin, E., Mérillou, S., and Benes, B. (2015). Environmental Objects for Authoring Procedural Scenes. *Computer Graphics Forum*, 35(1):296–308.

Gurin, E., Digne, J., Galin, E., and Peytavie, A. (2016a). Sparse representation of terrains for procedural modeling. *Computer Graphics Forum (Proceedings of Eurographics 2016)*, 35(2).

Gurin, E., Galin, E., Grosbellet, F., Peytavie, A., and Gnevaux, J.-D. (2016b). Efficient modeling of entangled details for natural scenes. *Computer Graphics Forum*, 35(7):257–267.

Hartsook, K., Zook, A., Das, S., and Riedl, M. O. (2011). Toward supporting stories with procedurally generated game worlds. In *2011 IEEE Conference on Com-*

putational Intelligence and Games (CIG'11), pages 297–304. IEEE.

Kelly, G. and McCabe, H. (2007). Citygen: An interactive system for procedural city generation. In *Fifth International Conference on Game Design and Technology*, pages 8–16.

Mateas, M. and Stern, A. (2002). A behavior language for story-based believable agents. *IEEE Intelligent Systems*, 17(4):39–47.

Misztal-Radecka, J. and Indurkhya, B. (2016). A blackboard system for generating poetry. *Computer Science*, 17(2):265.

Parish, Y. I. and Müller, P. (2001). Procedural modeling of cities. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 301–308. ACM.

Shaker, N., Togelius, J., and Nelson, M. J. (2016). *Procedural Content Generation in Games: A Textbook and an Overview of Current Research*. Springer.

Silva, P. B., Eisemann, E., Bidarra, R., and Coelho, A. (2015). Procedural content graphs for urban modeling. *International Journal of Computer Games Technology*, 2015:10.

Smelik, R., Tutenel, T., de Kraker, K. J., and Bidarra, R. (2010). Integrating procedural generation and manual editing of virtual worlds. In *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*, page 2. ACM.

Smelik, R. M., Tutenel, T., de Kraker, K. J., and Bidarra, R. (2011). A declarative approach to procedural modeling of virtual worlds. *Computers & Graphics*, 35(2):352–363.

Smith, A. M. and Mateas, M. (2011). Answer set programming for procedural content generation: A design space approach. *IEEE Transactions on Computational Intelligence and AI in Games*, 3(3):187–200.

Togelius, J., Champandard, A. J., Lanzi, P. L., Mateas, M., Paiva, A., Preuss, M., and Stanley, K. O. (2013a). Procedural content generation: Goals, challenges and actionable steps. *Dagstuhl Follow-Ups*, 6.

Togelius, J., Preuss, M., Beume, N., Wessing, S., Hagelbäck, J., Yannakakis, G. N., and Grappiolo, C. (2013b). Controllable procedural map generation via multiobjective evolution. *Genetic Programming and Evolvable Machines*, 14(2):245–277.

Treanor, M., Blackford, B., Mateas, M., and Bogost, I. (2012). Game-o-matic: Generating videogames that represent ideas. In *Procedural Content Generation Workshop at the Foundations of Digital Games Conference*.

UnityTechnologies (2016). Unity game engine. http://unity3d.com/unity. Consulted August 2016.

Verborgh, R., Van Deursen, D., Mannens, E., Poppe, C., and Van de Walle, R. (2012). Enabling context-aware multimedia annotation by a novel generic semantic problem-solving platform. *Multimedia Tools and Applications*, 61(1):105–129.