TU WIEN Informatics

Blickwinkelabhängige Surrogate-Terminale für Prozedurale Erzeugung von Geometrie

BACHELORARBEIT

zur Erlangung des akademischen Grades

Bachelor of Science

im Rahmen des Studiums

Medieninformatik und Visual Computing

eingereicht von

Moritz Roth
Matrikelnummer 01633060

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Univ.Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer
Mitwirkung: Univ.Ass. Chao Jia, BSc MSc
Univ.Ass. Dipl.-Ing. Dr.techn. Bernhard Kerbl, BSc

Wien, 4. Juli 2021

Moritz Roth | Michael Wimmer

TU WIEN Informatics

View-Dependent Surrogate Terminals for Procedural Geometry Generation

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Media Informatics and Visual Computing

by

Moritz Roth
Registration Number 01633060

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer
Assistance: Univ.Ass. Chao Jia, BSc MSc
Univ.Ass. Dipl.-Ing. Dr.techn. Bernhard Kerbl, BSc

Vienna, 4th July, 2021

Moritz Roth	Michael Wimmer

Erklärung zur Verfassung der Arbeit

Moritz Roth

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 4. Juli 2021

Moritz Roth

Kurzfassung

Prozedural erzeugte Geometrie spielt eine immer größer werdende Rolle in der Film- und Computerspieleindustrie. Shape Grammars haben sich als gängige Lösung für prozedurale Geometrie Generation im Bereich Architektur etabliert. Inzwischen sind diese so effizient, dass die Erzeugung von Gebäudegeometrie on Demand und in Echtzeit ermöglicht wird. Die laufend wachsende Nachfrage für immer schöner werdende Visualisierungen fordert jedoch die Weiterentwicklung von Shape Grammars. Komplexe Shape Grammars kommen schnell an die Limits der zur Laufzeit verfügbaren Ressourcen. Daher sind vereinfachende Maßnahmen gefordert, die jedoch gleichzeitig möglichst wenig Einfluss auf das Aussehen der erzeugten Geometrie haben sollen. Diese Arbeit untersucht wie blickwinkelabhängige Surrogate-Terminale in eine Shape Grammar integriert werden können und ob diese Vorteile mit sich bringen. Surrogate-Terminale beenden die Auswertung einer Shape Grammar frühzeitig und ersetzen feine Details mit im Vorhinein gerenderten Bildern. Wir legen eine Implementierungsmöglichkeit für blickwinkelabhängige Surrogate-Terminale fest und finden eine automatisierte Prozedur, die diese in eine Shape Grammar integriert. Unsere Lösung vergleichen wir anhand von drei Test-Szenarien mit einer unveränderten Shape Grammar und einer alternativen Methode. Testergebnisse zeigen, dass sich unsere Lösung von vorherigen Ansätzen unterscheidet, indem sie vermeidet unvollständige Geometrie zu erzeugen die eindeutig als solche erkennbar ist. Doch, obwohl unsere Methode in großen Szenen um einiges schneller ist als die unveränderte Shape Grammar, sind vorherige Ansätze noch effizienter. Wir kommen zum Schluss, dass blickwinkelabhängige Surrogate-Terminale vielversprechende Ergebnisse liefern. Jedoch sind weitere Verbesserungen der Methode notwendig, um sie performancetechnisch auf den gleichen Stand zu bringen wie bereits existierende Herangehensweisen.

Abstract

Procedural geometry generation plays an ever-increasing role in the movie- and video gaming industry. Shape grammars have established themselves as the preferred solution for procedural architecture generation. Research in past decades drastically improved the speed of geometry derivation through shape grammars, making it possible to generate 3D buildings on-demand and in real-time. However, the constantly rising demand for high-quality visualizations requires new measures to reduce complexity in 3D models generated by shape grammars without sacrificing visual quality. This thesis explores the feasibility and benefits of inserting view-dependent surrogate terminals into a shape grammar. Surrogate terminals end grammar derivation early and approximate finer details with pre-rendered images. We find a possible solution for implementing view-dependent surrogate terminals and describe a scheme to automatically insert them into a shape grammar. Results show that contrary to previous approaches, our method avoids the generation of visibly incomplete geometry. However, even though the modified shape grammars evaluate faster than the original in large scenes, previous methods provide a more significant performance gain. We conclude that view-dependent surrogate terminals provide promising results, but further optimization is necessary to match the performance of prior techniques.

Contents

CHAPTER 1

Introduction

As the scope of 3D video game projects and movies increases, it is becoming less and less feasible to craft required assets by hand [HMVI13, Ios11]. Over the past decades, various procedural generation techniques have been developed to automatically create different elements such as sounds, images, effects, vegetation, urban environments, puzzles, and even levels [HMVI13]. Adoption of such techniques in mainstream games is reportedly slow due to many of the developed solutions not being considered general-purpose [HMVI13]. However, primarily games that feature huge virtual worlds, such as Minecraft, No Man's Sky, and Elite Dangerous, tend to employ procedural generation techniques [TN21, HMVI13, Str15].

Procedural geometry generation uses rules to define 3D objects implicitly rather than storing them explicitly. There are multiple benefits to doing this: Representing highly detailed geometry through rules is more compact, as thousands of vertices can be described with a few rules. Furthermore, rules can be evaluated dynamically, react to various circumstances, and adjust the generated geometry accordingly [MWH+06]. I.e., less detailed geometry is produced on low-end devices, or context-sensitive rules avoid intersection with other geometry in the same scene. In addition, deliberate variation of input parameters can lead to different outputs, making the generation of similar but not identical assets easy. Lastly, iteration on already generated assets is straightforward, as even large-scale changes can be realized by merely adjusting the rules and reevaluating them.

Architecture, in particular, is a prime candidate for procedural geometry generation. Urban environments are frequently featured in movies and video games, often being a point of interest or even central to the experience. Such environments typically contain vast amounts of unique buildings. Hence, crafting these assets by hand does not scale well. Buildings on their own use repeating patterns [MWH+06] and tend to conform to architectural design principles, local building regulations, or an overall art direction.

That makes it easier to define firm sets of rules that generate various buildings given only a handful of input parameters.

As mentioned above, the film- and gaming industry profit the most from procedural architecture since fictional cities can be created and iterated upon in a scalable fashion. However, procedural architecture generation is also helpful in non-fictional scenarios. Unfortunately, highly detailed 3D models of buildings and infrastructure in the real world are rarely readily available [SLG19]. Therefore, a common approach is to use whatever data is available to generate a fairly accurate representation. One example of a video game using procedural architecture to model real-world urban environments can be found in Microsoft Flight Simulator 2020 [JF20]. Other use-cases include urban planning- and visualization applications, which are subject to the same scarcity of available building geometry.

Shape grammars have established themselves as viable solutions for efficiently generating coherent, context-sensitive 3D buildings while also providing artistic control [MWH+06, WWSR03]. A set of rules is used to derive building geometry from an initial shape. Optimization of this derivation process has advanced to a point where it is possible to generate urban environments on-demand in real-time on the Graphics Processing Unit (GPU) alone. This skips the comparably slow process of loading big chunks of geometry data onto the GPU for rendering [SKK+14a, SKK+14b]. As a trade-off, geometry derivation speed now becomes a relevant factor, and any post-processing techniques that reduce geometry complexity also need to run in real-time. Traditionally, having too much geometry in a scene is avoided by preparing different versions of assets (levels of detail (LODs)) with decreasing complexity. When sending data to the GPU, an adequate version is selected to keep the scene's overall complexity in check. Dynamic generation of different LOD geometry during grammar derivation reduces both geometry complexity and derivation time [SKK+14b]. Previous work by Steinberger et al. [SKK+14b] suggested the insertion of pre-computed surrogate terminals into a shape grammar. These surrogate terminals insert pre-rendered images of building parts into the scene, replacing the actual parts of the building they represent. This works particularly well for windows and finer details in facade decoration. For balconies or similar parts protruding out of a building's surface, this is a less optimal solution since a single image can only represent the geometry from one viewing direction.

This thesis introduces view-dependent surrogate terminals (VDSTs). The idea is to select one out of multiple pre-rendered images (surrogate textures) depending on the direction from which the camera views the geometry in question. Since architecture in urban environments is the most common use-case for shape grammars, we have decided to limit the scope of this thesis to procedural generation of urban architecture. We explore the benefits and drawbacks of VDSTs and aim to find out if they can easily approximate non-planar structures that significantly protrude from the original shape. Another question is whether VDSTs increase the speed of derivation and rendering due to early termination and reduction of overall geometry.

Prior techniques and other work relevant to our method are summarized in Chapter 2. Throughout Chapter 3, we describe our implementation of VDSTs and how we insert them into a shape grammar. Chapter 4 compares our method to the unmodified shape grammar and previous work based on three examples. Limitations and future work are discussed in Chapter 5. Finally, in Chapter 6, we shortly sum up our work and answer the research questions.

CHAPTER 2

Related Work

Explicit research on our selected topic, i.e., optimizing the generation of procedural geometry with simplified geometry, is rare. Therefore, we first explain procedural architecture generation through shape grammars in Section 2.1. Geometry optimization in the form of image-based impostors and visibility culling is explored in Sections 2.2 and 2.3. Then, Section 2.4 summarizes Image Quality Assessment (IQA) techniques. Finally, in Section 2.5, we describe the method proposed by Steinberger et al. [SKK+14b] that serves as a reference solution for this work and uses image-based impostors, visibility culling, and IQA to enhance shape grammars.

2.1 Shape Grammars & Procedural Architecture

Stiny et al. [SG71] first proposed the idea of a grammar that operates on shapes instead of symbols. Their proposal refers to a shape as a set of one or more geometric objects in a specific configuration within 2D / 3D Euclidean space. They redefine the Kleene operator * for sets of shapes: For any set V that exclusively contains shapes, V^* is the set of all possible combinations of the shapes within V, where each shape can appear an arbitrary number of times **and have an arbitrary scale and orientation each time it appears**. A shape grammar consists of

- a finite set of terminal shapes V_T
- a finite set of marker shapes V_M, which are distinct from V_T, meaning $V_T^* \cap V_M = \emptyset$
- a finite set of rules R
- an initial shape $I \in V_T^* \times V_M$

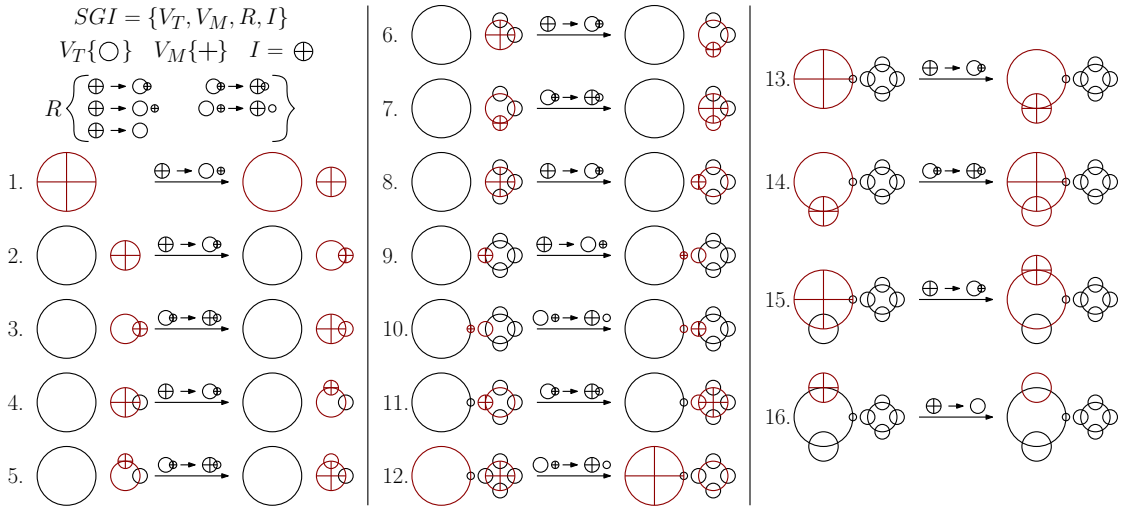


Figure 2.1: Example of a shape grammar (top-left) together with one possible step-by-step derivation. The notation above the arrow for each step indicates which rule is being applied. The left- and right-hand sides of the arrow show the shape before- and after rule application, respectively. Shapes directly manipulated in the current step are highlighted in red.

Together they form the 4-tuple $SG = (V_T, V_M, R, I)$, which defines the shape grammar. Rules have a left-hand side u and a right-hand side v where

$$u = t_1 m_1, v = \begin{cases} t_1 & |case^1 \\ t_1 m_2 & |case^2 \\ t_1 t_2 m_2 & |case^3 \end{cases}, t_i \in V_T^*, m_i \in V_M$$

and are denoted as $u \to v$. The arrow signifies that u can be replaced with v. Algorithm 2.1 shows how to derive a shape grammar. Figure 2.1 shows an example grammar together with its derivation.

While the shape grammar by Stiny et al. [SG71] provided means to model and analyze different kinds of art and architecture, it was still mainly derived by hand and lacked coherent notation. The combination of different rules often resulted in unwanted side effects, and deducing the impact of a single rule on the derived shapes proved difficult [MWH+06]. Follow-up papers of Wonka et al.[WWSR03], Marvie et al. [MBG+12], and Müller et al. [MWH+06] addressed these issues and proposed new methods as well as modifications to the model.

Müller et al. [MWH+06] proposed to link shapes to terminal- (Σ), or non-terminal (V) symbols $v \in \Sigma \cup V$ and made it possible to notate a shape grammar in text form. Each shape additionally keeps track of its size S and orientation by storing a position P together with vectors X, Y, and Z forming the local coordinate system. Finally, a shape stores a set of grammar-specific parameters p, making it a 3-tuple $s = (v, g, p); g = (P, X, Y, Z, S); P, X, Y, Z, S \in \mathbb{R}^3$.

Algorithm 2.1: Stiny shape grammar derivation

Input: $SG = (V_T, V_M, R, I)$

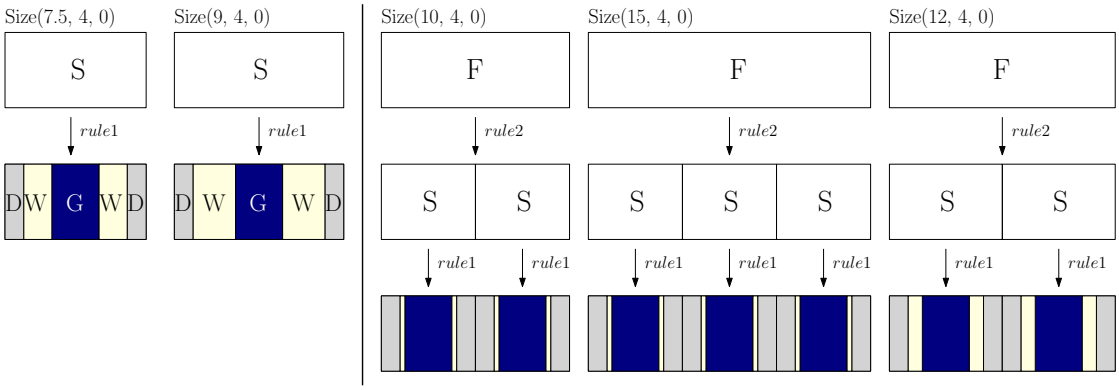
$S_0 = I$;
$n = 0$;
while *true* **do**
  $matching\_shapes = \emptyset$;
  for $r_i = (u_i, v_i)$ *in* R **do**
    find all $u_{i_k} \in S_n$ in any orientation or scale;
    append all found u_{i_k} to $matching\_shapes$;
  if $matching\_shapes == \emptyset$ **then**
    break;
  select one u_{i_k} from $matching\_shapes$;
  $S_{n+1} = (S_n \setminus u_{i_k}) \cup v_{i_k}$;
  $n = n + 1$;
return S_n;

Production rules are restricted to only have a single non-terminal symbol $v \in V$ on the left-hand side. This makes the difference between the shape grammar by Stiny et al. [SG71] and the shape grammar by Müller et al. [MWH+06] analogous to the difference between context-sensitive and context-free grammars. In context-sensitive grammars, the left-hand side of a production rule may include multiple non-terminal and terminal symbols. On the other hand, the left-hand side of a rule in a context-free grammar is only one non-terminal symbol. This change streamlines the derivation process and makes the effect a rule has on the outcome more apparent.

A unique id now identifies rules. The left-hand side $v \in V$ is combined with a condition. The right-hand side is a parameterized set of operations to be executed upon any shape with symbol v to compute the successor shapes, combined with a selection probability [MWH+06]. The following notation is used:

$$id : non\text{-}terminal : condition \rightsquigarrow operations : probability$$

Operations include basic instructions such as affine transformations. However, Müller et al. [MWH+06] also introduced operations Subdivide and Repeat inspired by common patterns found in architecture. Subdivide splits the original shape along a local axis into a predetermined set of sub-shapes and balances the sizes of the derived shapes to fit the size of the original shape (Figure 2.2 left). This is akin to refining a coarse structure by adding more detailed structures in a specific pattern. Repeat creates sub-shapes with predetermined size along a local axis within the original shape. It produces the appropriate number of sub-shapes to fill the original shape and stays within its bounds (Figure 2.2 right). This facilitates tiling a repeatable pattern across shapes of different sizes.



$1 : S \rightarrow SubDiv(\text{"}X\text{"}, 1, 1r, 2.5, 1r, 1)\{D|W|G|W|D\}$
$2 : F \rightarrow Repeat(\text{"}X\text{"}, 5)\{S\}$

Figure 2.2: *Left*: Subdivide operation. Symbols D and G are defined with absolute width, ensuring that the sub-shapes always have the same size regardless of parent-shape size. Symbols W are defined with relative width (marked by an r), meaning they will adjust their size to fill the remaining space. *Right*: Repeat operation. Shape F is replaced by multiple instances of shape S with width 5. If an exact fit is not possible, the width for all sub-shapes is adjusted slightly to avoid one out-of-place sub-shape.

More recent advancements in the field of shape grammars are mainly concerned with increasing the performance of shape grammar derivation. For example, Steinberger et al. [SKK+14a] describe a rule scheduling scheme that, especially when implemented to work on the Graphics Processing Unit (GPU), achieves enormous speedups. In a follow-up paper, Steinberger et al. [SKK+14b] use image-based impostors, visibility culling techniques, and IQA to further shape grammar performance and make real-time geometry derivation possible. The following three Sections, 2.2, 2.3, and 2.4, give a broad overview of the concepts mentioned above before Section 2.5 describes Steinberger's solution in more detail.

2.2 Image-Based Impostors

Over the past few decades, demand has been on the rise for 3D scenes of ever-increasing size, complexity, and geometric detail. While technical advancements of the underlying hardware regularly boost the performance of commercially available computers, they struggle to keep up with the workload [JWP05]. Thus other solutions are required. A common approach is to reduce unnecessary geometric detail that only appears very small in the rendered image. The term level of detail (LOD) describes a technique where multiple versions of the same asset with decreasing quality (vertex-count, texture resolution, etc.) are provided. Less qualitative versions (impostors) are used when the asset is further away or not the center of attention, and the difference in quality is less likely to be noticed. This effectively reduces the number of vertices in a scene and helps

to alleviate aliasing effects that would otherwise occur due to faces being smaller than pixel size [DDSD03]. The following paragraphs describe multiple approaches to create different LOD assets from detailed geometry automatically.

Billboards and sprites replace complex geometry with planar impostors. More specifically, they are rectangles textured with an RGBA image of the geometry. Since one image can approximate arbitrarily complex geometry without requiring additional resources, they are a very light-weight and efficient method. The textures for sprites and billboards can be created manually from images of real-world objects. However, textures may also be generated automatically by rendering the complex geometry that is supposed to be approximated.

The exact differences between sprites and billboards are vaguely defined and vary between publications. In the context of the work by Jeschke et al. [JWP05], the terms are synonymous. Décoret et al. [DDSD03] avoid the expression sprite but define a billboard as either two planar impostors intersecting each other, forming a cross or a single impostor that aligns itself always to stay parallel to the image plane. Eberly [Ebe01] makes a strict distinction between sprites and billboards. Sprites are planar impostors with arbitrary orientation. Billboards are a special case of sprites that are in some form oriented towards the camera. Eberly discerns between *screen-aligned* billboards that stay parallel to the image plane and remain upright regardless of camera rotation and *axially aligned* billboards that only rotate about their up-vector in object space. For the purposes of this thesis, we will stick to the definitions of Eberly.

One drawback of sprites is that they only look similar to the original geometry if viewed from roughly the same camera position as the one used to generate their texture. If the discrepancy gets too big, the sprite starts to look unnatural and out of place (see Figure 2.3). In that case, another better-fitting sprite has to be rendered or selected from a set of pre-rendered sprites. Alternatively, the detailed geometry can be displayed. Since the difference between sprite and original diminishes with increasing view distance, the region in which a sprite stays valid is defined by a minimum distance, the direction from which the sprite was rendered, and an angle that defines an acceptable amount of visual inaccuracy (see Figure 2.4) [JWP05].

Another drawback of billboards and sprites is that the z-buffer information is lost when reducing geometry into a rendered texture. Because of this, visibility errors can occur when intersecting with other objects in the scene or when two planar impostors should occlude each other (see Figure 2.5). In static scenes, it is possible to circumvent this issue by grouping objects close to each other in a single sprite, but this is no longer possible in a dynamic scene, where objects can move independently. Schaufler [Sch97] proposes nailboards that store a Δ-value in the alpha channel of their texture to address this issue. This value is indicative of the z-buffer difference between the nailboard and the original geometry and makes it possible to fix the visibility errors. Since the alpha channel was previously used to determine the transparency of the texture, this method defines one explicit alpha value to signify that a given texel is transparent. The rest of the values imply that the texel is opaque and encode the z-buffer difference.

(a) Scene Overview (b) Generating Position (c) Valid Position (d) Invalid Position

Figure 2.3: While sprites stay valid for similar camera positions, they start looking more and more out of place as the camera is moved away from the position the sprite was generated from. This effect is much more apparent for camera movement around the object than for movement that changes the distance between camera and sprite. We generated a sprite of a wet floor sign from camera position (b) to demonstrate this effect. A second camera position (c) is close to (b), while a third camera position (d) is farther away from (b) and views the sprite from a different angle. The sprite looks acceptable in the image rendered from camera position (c) but looks out of place in the image rendered from camera position (d).

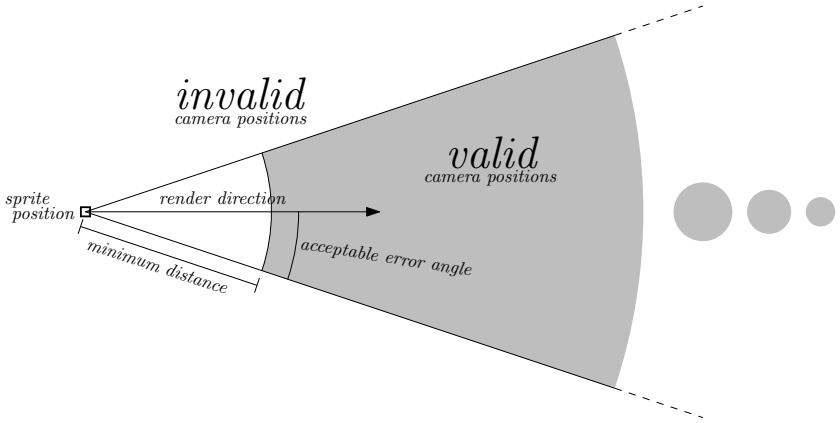


Figure 2.4: Space can be divided into sets of valid- and invalid camera positions for a given sprite based on the direction the sprite was rendered from, a minimum distance, and the maximal tolerable deviation angle from the original render direction [JWP05].

(a) original geometry (b) chair in front of desk (c) desk in front of chair

Figure 2.5: In some areas, the chair should be displayed in front of the desk, while in others, it should be behind the desk. Since the z-buffer information of the geometry is being reduced to the face of the billboard, those intricate interactions cannot be reproduced. It is only possible to position one billboard in front of the other or let the planes intersect on a straight line [Sch97].

In later work, Schaufler [Sch98] proposed another approach. Here the impostor texture is created by rendering the inside of the geometry from behind using a perspective projection. This way, a single render can display more sides of the original geometry than a conventional render from the outside could show. In order to render the impostor, multiple sprites are stacked along the z-axis of the view frustum that was previously used to create the impostor. Each rectangle exclusively displays those fragments of the impostor texture with a z-buffer value between the z-distance of the rectangle itself and the z-distance of the next rectangle. Thus, the 3D features of the original geometry remain intact, and the impostor remains valid in a broader range of camera positions. Figure 2.6 left demonstrates how the impostor texture is generated. Figure 2.6 right shows how the impostor is rendered.

The method proposed by Décoret et al. [DDSD03] selects panes in 3D space that roughly tangentially align with as many triangles as possible. They then project the triangles onto these planes, creating textures for a cloud of semi-transparent sprites. This way, the complexity of the geometry can be drastically reduced, and the impostor remains valid for all camera angles. In order to adjust to different LOD, an error threshold can be adjusted that influences how many sprites will be created.

2.3 Visibility Culling

Other common techniques that reduce render expense can be found in the field of visibility culling. These approaches distinguish between visible and non-visible geometry and discard non-visible geometry early on so that no resources are wasted on it [JWP05,

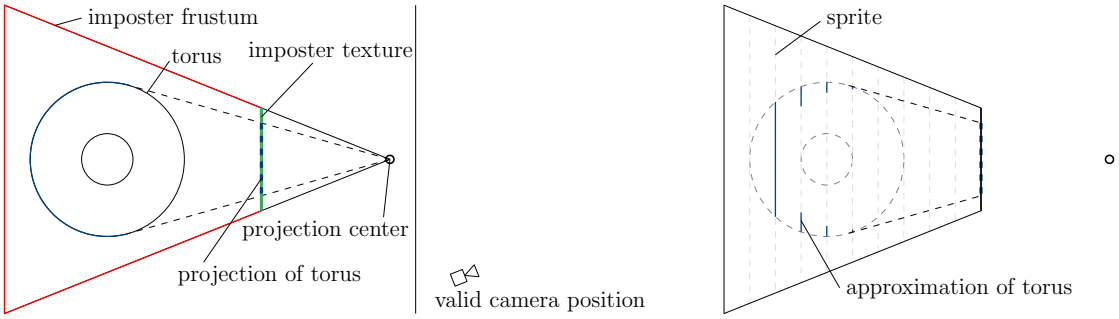
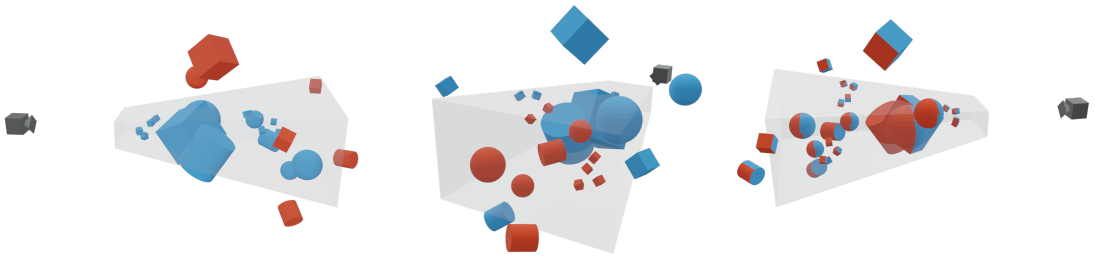


Figure 2.6: Left: The inside of the object (in this case a torus) is rendered to an impostor texture. Note that in this render-pass, front faces are being culled, and fragments with greater distance to the camera are drawn over closer fragments. Right: Multiple layered sprites use the impostor texture to more accurately represent the original geometry [JWP05].



(a) Frustum culling (b) Occlusion culling (c) Back-face culling

Figure 2.7: Different visibility culling techniques. Rendered geometry is shown in blue, discarded geometry is shown in red.

CCSD03]. Unlike image-based impostors, since only non-visible geometry is removed, the visual appearance remains exactly the same as without visibility culling. The efficiency of such an algorithm is determined by its accuracy and the added computational overhead. In general, there are three types of visibility culling: Frustum Culling, Occlusion Culling, and Back-face Culling. Frustum culling checks if objects can be visible inside the render by comparing their geometry with the camera's viewing frustum. If an object is entirely outside the viewing frustum, it cannot be visible in the rendered image and is therefore discarded (see Figure 2.7a). Occlusion culling finds geometry entirely hidden by other geometry from the current camera position and discards it (see Figure 2.7b). Finally, back-face culling does not draw faces that do not face the camera (see Figure 2.7c).

Back-face culling, being the most straightforward and simple approach, can be used in the broadest range of applications. It is easy and quick to determine whether a face

is pointing away from the camera since only the face normal needs to be compared with the vector from the camera to the face position using a dot product [Ebe01]. It is part of the OpenGL rendering pipeline and, therefore, usually does not have to be implemented separately. Typically, scenes consist of objects with faces pointing in very different directions, making it possible for back-face culling to cull a significant amount of faces regardless of camera position.

Frustum culling is more complex than back-face culling. In order to check whether an element is within the view frustum, it is compared with the frustum's six sides. This makes it no longer efficient to check each face or even each object. Because of this, a bounding volume usually approximates one or more objects. Different approaches have been proposed using axis-aligned bounding boxes, oriented bounding boxes, or bounding spheres [AM00]. Some methods use hierarchical scene partitioning techniques to cover big chunks of a scene at once when making a comparison [Chi95, AM00]. Regardless of all efforts made to increase the performance of frustum culling, there are still cases in which the computational overhead of the procedure overshadows the benefits. Suppose all of the scene is contained within the view frustum. No object could be culled, but it would still be necessary to make at least a few comparisons.

Occlusion culling only improves performance under the condition that much geometry is actually being occluded, e.g., the camera is placed on ground level in a dense city. In this case, close-by buildings would occlude most of the remaining scene. On the other hand, if the camera were placed in a bird's-eye view to give a good overview of the city, much less geometry would be occluded, reducing the effectiveness of occlusion culling to a point where it reduces overall performance [SKK+14b]. Checking if objects are occluded is much more complex than applying frustum- or back-face culling since occlusion is a property dependent on the camera, the object itself, and all the other objects within the scene [CCSD03].

2.4 Image Quality Assessment

Objective Image Quality Assessment (IQA) tries to determine the quality of images algorithmically. While some approaches find a quality score based on intrinsic properties of an image, others rely on the existence of a reference image, to which they compare the tested image [MES14]. A straightforward method (per-pixel difference) calculates the difference for each pixel in two images using the mean value, which makes it fast and easy to implement. However, in most cases, this method fails to adequately capture how humans see images and is therefore unfit to represent the visual difference perceived by humans. For example, an image compared to itself using per-pixel difference would yield a drastically different result than the same image compared to a version of the image shifted a few pixels to the right or a version of the image with a slightly different tint. While these changes would be hard for a human to notice, they significantly impact the result of the per-pixel comparison.

Structural Similarity (SSIM) attempts to better approximate human vision by comparing weighted k-by-k pixel sub-areas of the image, which later get summarized through a mean value. The method only considers the luminance values of each image. It combines the quality score for each k-by-k window out of three metrics, comparing the weighted average luminance, the weighted average contrast (standard deviation), and weighted structural differences (correlation) [WBSS04, MES14, VWB+21].

While SSIM handles the above-mentioned issues much better than per-pixel difference, the fixed sub-area size acts as a limiting factor. As a result, the method mainly detects frequency differences that can be captured within the k-by-k window. Wang et al. [WSB03] proposed Multi-Scale Structural Similarity (MS-SSIM) to address this issue. They factor in multiple versions of the image, each successive version being the previous image filtered with a low-pass filter and downsampled to half the resolution.

The method Butteraugli, introduced by Alakuijala et al. [A+, AOS+17], considers color values and factors in quirks of human vision that make it easier or harder to notice certain differences. They take into account that we perceive color differences with varying accuracy due to different proportions of color-sensing cones in our retina. Further, they account for the fact that it is harder to notice differences between images in parts with high frequencies than parts with lower frequencies.

2.5 Reference Solution

As mentioned in Section 2.1, the solution proposed by Steinberger et al. [SKK+14b] improved shape grammars with the use of image-based impostors, visibility culling, and IQA. A pre-processing step is introduced that inserts surrogate terminal operations into a shape grammar. These surrogate terminal operations replace detailed geometry with a pre-rendered sprite of said geometry if the distance to the camera is big enough. This effectively reduces not only vertex-count but also derivation time since derivation is terminated early. To find the appropriate distance thresholds that decide whether a surrogate terminal can be used, Steinberger et al. render both the detailed geometry and the surrogate terminal from multiple camera angles and distances. They then fit a parameterized sigmoid function to the correlation of the camera distance with the per-pixel difference of the rendered images and compute the distance threshold. Figure 2.8 shows the geometric difference that such a surrogate terminal has on the final geometry.

To further increase performance, frustum- and occlusion pruning is employed [SKK+14b]. Visibility culling makes sure that invisible geometry is discarded early on, whereas visibility pruning ensures that invisible geometry is not even generated in the first place. To make frustum pruning possible, coarse bounding volumes, also called *hulls*, are calculated dynamically during derivation. When designing a shape grammar, certain rules are tagged as hull-generating rules. Only those rules are used to calculate a bounding volume, and all child rules are assumed to generate geometry contained within the hull. Once the hulls have been determined, they are checked against the camera's viewing frustum to decide if further derivation is required. Similar measures are required for

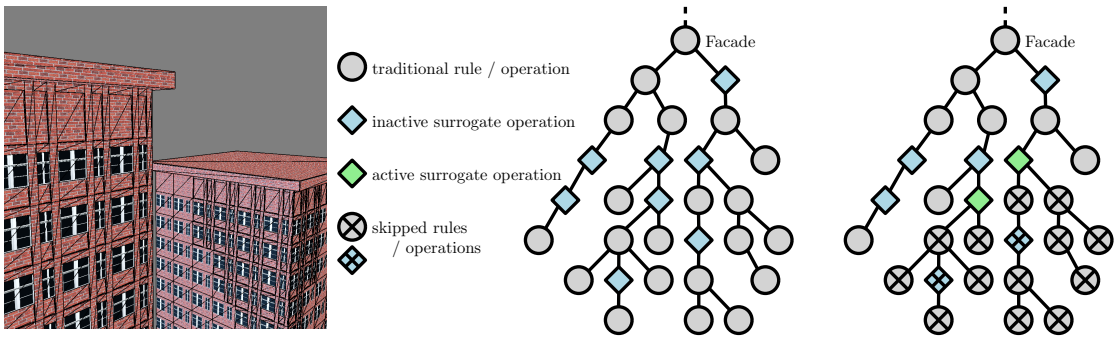


Figure 2.8: Left: Derived geometry from a shape grammar utilizing surrogate terminals. Nearby structures are generated in full detail, while distant facade elements are replaced with sprites. Right: Two excerpts of a derivation tree (inexact and simplified representations of the left image) that demonstrate the general concept of surrogate terminals.

occlusion pruning. However, the bounding volumes need to be more accurate to avoid pruning geometry that is actually visible through a hole in some otherwise occluding geometry.

Together with the more efficient rule scheduling scheme proposed in prior work by Steinberger et al. [SKK+14a], these advancements allow for generating and rendering entire city visualizations in real-time on the GPU alone. In our work, we aim to improve upon the surrogate terminal operations introduced by Steinberger et al. and further use the method as a benchmark to compare our solution against.

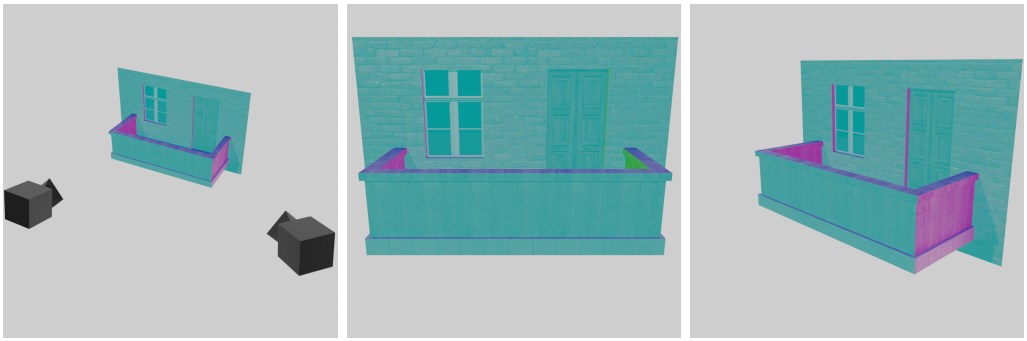
CHAPTER 3

Methods

This thesis is a follow-up work to Steinberger's On-the-fly Generation and Rendering of Infinite Cities on the GPU [SKK+14b]. In their paper, Steinberger et al. propose a method to skip multiple derivation steps inside a shape grammar and reduce unnecessary geometric detail by inserting additional surrogate terminal operations. Those surrogate terminals essentially replace detailed geometry with pre-rendered sprites [JWP05] of said geometry. The grammar chooses to use them if the observing camera is far enough away so that the Human Visual System (HVS) can barely notice the difference. Our proposed method builds on the insight gained in this work and aims to improve surrogate terminals by broadening their application domain.

According to Steinberger et al. [SKK+14b], one of the main issues with generating surrogate textures is the discrepancy in dimensionality between input shape and final geometry. The input shape is an arbitrary non-terminal shape, and the final geometry is its entirely derived counterpart. A two-dimensional quad, for example, could be used as a baseline to generate detailed geometry for a window. It could, however, also be used as a building footprint that will subsequently be extruded into a 3D prism. In the first case, all of the generated geometry stays relatively close to the input shape, making it easy to render a sprite that stays valid for a wide range of view directions and distances. In the second case, some of the geometry is very far away from the input shape. The distant geometry is subject to the parallax effect. This limits the range of valid view directions severely and increases the minimum distance from which it makes sense to use the surrogate terminal instead of the actual geometry [JWP05].

We observe that the range of valid view directions for a given sprite is limited in two ways, both caused by the parallax effect: First, vertices distant from the sprite are impacted differently to close vertices when the camera moves. Second, some faces are only visible from certain directions [JWP05], as demonstrated in Figure 3.1. The effects of the first limitation loom larger with greater distance between the geometry to the impostor and diminish with increasing camera distance. However, the effects of the second limitation

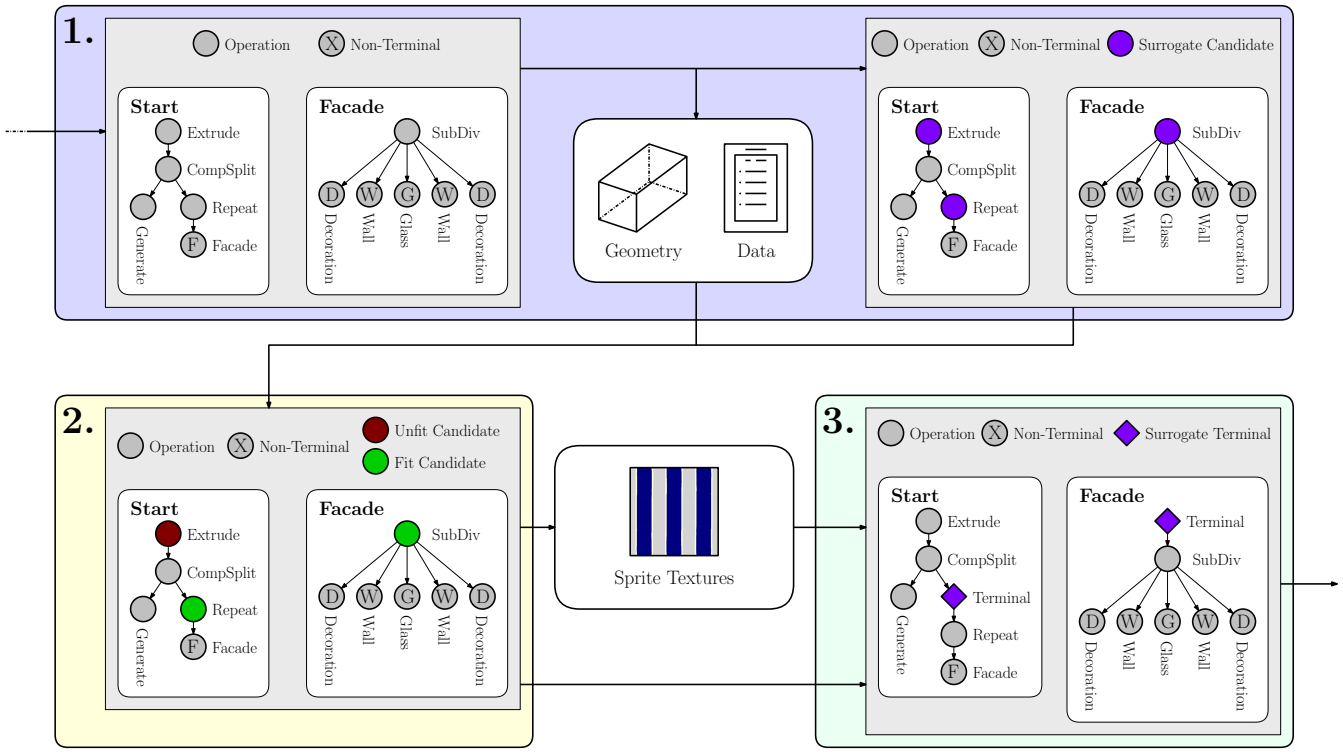


(a) Overview (b) Front view (c) Side view

Figure 3.1: The right face of the balcony (magenta) is visible from the side while hidden from the front. Because of this, it is not possible to create a valid sprite that can be used for both views.

persist regardless of camera distance and are even apparent in cases where the geometry is not that far away from the sprite. Consequently, if the generated geometry is farther away from the input shape, it is impossible to use a single sprite to cover a wide range of viewing angles. However, it would still be desirable to stop grammar derivation early to avoid wasting computation time on unnecessary details. In architecture, it is not uncommon for geometry to stick out of an otherwise flat surface. Balconies, vents, pipes, satellite dishes, and emergency stairs are only some examples of geometric detail an architecture-oriented shape grammar may try to model. Steinberger's [SKK+14b] method cannot provide satisfactory sprite textures in these cases. Our proposed approach addresses this issue using multiple, pre-rendered sprite textures for different points of view selected on a case-by-case basis once the grammar is executed.

Just like Steinberger et al. [SKK+14b], we propose a pre-processing step that transforms an unmodified version of the grammar (original grammar) into a modified version (modified grammar). The modified grammar utilizes surrogate terminals, while original grammar does not utilize surrogate terminals. First, we build an Abstract Syntax Tree (AST) of the original grammar, which we use to derive and evaluate the grammar once from top to bottom. In the derivation process, we identify operations of certain types as candidates for surrogate terminal insertion (surrogate candidates). For these surrogate candidates, we remember which parts of the geometry they create and the input shape they used to create that geometry. In a second step, we analyze all found surrogate candidates based on their final geometry and input shape and determine whether they can be used. Next, we render multiple surrogate textures from different camera configurations for each suitable surrogate candidate. Finally, we insert the rendered surrogate textures as surrogate terminal operations into the AST and translate the modified AST to get the modified grammar. Figure 3.2 gives a visual overview of the individual stages and demonstrates how a grammar is modified based on a simple example.



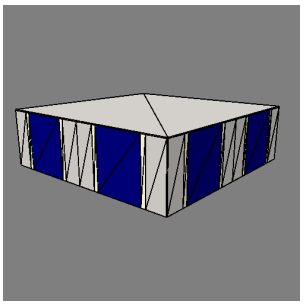
(a)

```
Start:
  Extrude(3)
  CompSplit() {
    mantle:
      Repeat(+X, 5)
      Facade;
    default: Generate;
  };

Facade:
  SubDiv(+X) {
    1.0 : Decoration;
    1.0r: Wall;
    2.5 : Glass;
    1.0r: Wall;
    1.0 : Decoration;
  };

Wall:
    ...

...
```

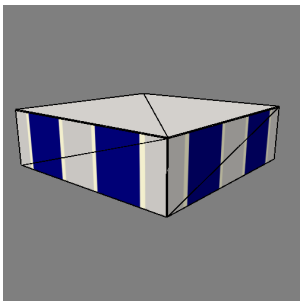
(b)

```
Start:
  Extrude(3)
  CompSplit() {
    mantle:
      Terminal(...)
      Repeat(+X, 5)
      Facade;
    default: Generate;
  };

Facade:
  Terminal(...)
  SubDiv(+X) {
    1.0 : Decoration;
    1.0r: Wall;
    2.5 : Glass;
    1.0r: Wall;
    1.0 : Decoration;
  };

Wall:
    ...

...
```

(c)

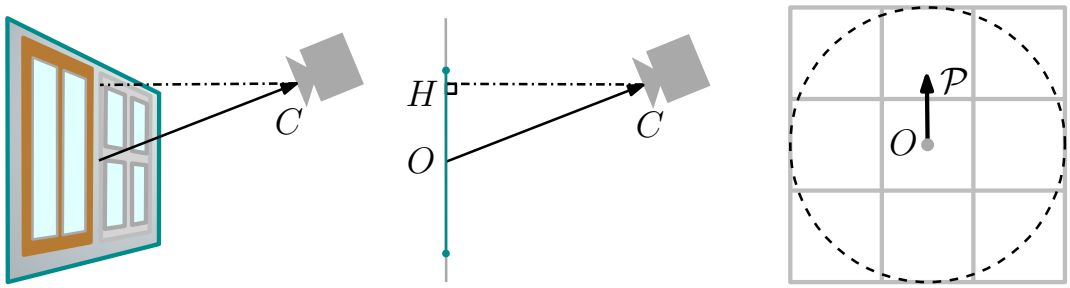


(d)



(e)

Figure 3.2: (a) Overview of our method. (b) Example: original grammar. (c) Example: modified grammar. (d) Render: original grammar. (e) Render: modified grammar.



(a) (b) (c)

Figure 3.3: Visual demonstration of how to calculate the point of view indicator (POV indicator) $\mathcal{P}$.

Section 3.1 describes how exactly our novel, view-dependent surrogate terminals operate once inserted into the grammar. The following sections, 3.2, 3.3, 3.4, and 3.5, explain how to automatically generate surrogate terminals and insert them into a shape grammar. Section 3.6 then describes further improvements we added to make our solution more feasible.

3.1 Surrogate Terminals

The surrogate terminal operations proposed by Steinberger et al. [SKK+14b] work only on planar input shapes. They decide whether to derive the grammar further or insert a sprite based on camera distance alone. We augment these surrogate terminals to consider the camera's position relative to the surrogate and select the most appropriate from multiple sprite textures. To facilitate this, a suitable metric is required, which needs to:

- capture the direction from which the camera observes the input shape
- be quick to compute
- allow the best fit surrogate texture to be selected with few comparisons

We propose a novel point of view indicator (POV indicator) $\mathcal{P}$, which is a 2D vector with length ≤ 1. To obtain $\mathcal{P}$, we first calculate the vector $\overrightarrow{OC}$ from the input shape origin O to the camera position C inside object space of the input shape (see Figure 3.3). Then, we project $\overrightarrow{OC}$ along the input shape's normal vector onto the input shape to get $\overrightarrow{OH}$, which we finally normalize by dividing by $\|\overrightarrow{OC}\|$ (see Equation 3.1).

$$\mathcal{P} = \frac{\overrightarrow{OH}}{\|\overrightarrow{OC}\|} \tag{3.1}$$

Doing these calculations in object space helps because the following shortcuts can be taken:

1. $\overrightarrow{OC} = C$, since $O = (0, 0, 0)$
2. projecting $\overrightarrow{OC}$ onto the input shape is as simple as multiplying one vector component by 0
3. converting $\mathcal{P}$ from a 3D vector to a 2D vector can be done by omitting the same vector component

$\mathcal{P} = (0, 0)$ if the camera is directly in front or behind the initial shape. Suppose the camera position lies on the same plane as the initial shape. In this case, $\|\mathcal{P}\|$ would equal 1. Overall, $\|\mathcal{P}\|$ quantifies how "head-on" the camera looks at the surrogate. The vector $\mathcal{P}$ itself captures the direction from which the camera views the shape.

We use the POV indicator to subdivide the space into multiple cells with similar values (see example in Figure 3.3c) and generate suitable surrogate textures for each cell. Once the surrogate terminal operation gets called, it calculates camera distance and POV indicator. Next, it compares the camera distance with a pre-computed distance threshold τ_D and decides whether to terminate derivation. If the decision is to derivation, an appropriate sprite is inserted into the scene as a terminal shape. The sprite's texture is decided by looking up in which cell of the subdivided POV indicator space $\mathcal{P}$ lies. If the camera is too close, derivation does not stop, child operations are executed, and the entire geometry is generated.

3.2 Surrogate Candidate Selection

In order to analyze and modify a shape grammar, we first need to build an AST from the grammar's notation. Each rule has its own tree, and operations form the tree nodes (see Figure 3.2a top-left). Leaves of a tree either discard unnecessary shapes (*Discard* keyword), add the current shape to the final geometry (*Generate* keyword), or assign a new non-terminal symbol to the shape.

Once the AST is complete, it can be used to derive the grammar by defining an axiom (tuple of initial shape and starting symbol) and recursively applying rules until no non-terminal shape remains. While navigating through the tree structure of the AST, we keep track of possible surrogate candidates using a separate data structure. Namely, each time we derive a possible surrogate candidate, we add a 4-tuple containing

1. a reference to the operation
2. the current shape $s = (v, g, p)$
3. the size of the global index buffer *before* the operation has been executed
4. the size of the global index buffer *after* the operation *and all its children* have been executed

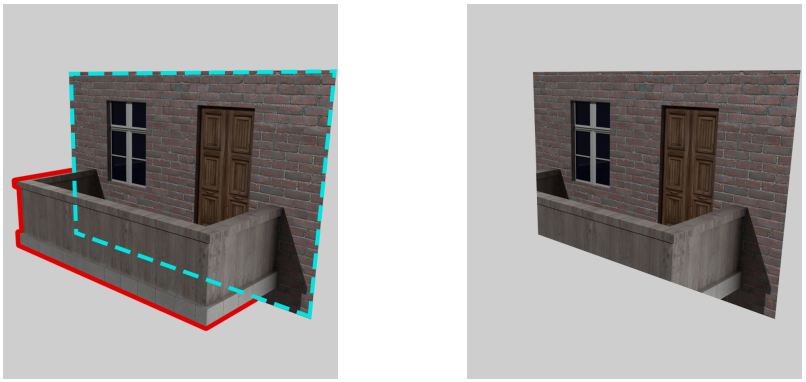
to a list. Because we derive the AST in a depth-first fashion and insert new indices at the end of global buffers, these index buffer offsets can later be used to analyze and exclusively render geometry created by the operation and its children.

Not every operation type makes a good surrogate candidate. The translate operation, for example, changes the position of the final geometry, while conditionals dynamically decide which children to invoke. Since the goal is to reduce geometric detail, creating surrogate terminals for operations that directly contribute to adding more geometry to the scene makes sense, for example, operations that subdivide, duplicate, or extrude the input shape.

3.3 Surrogate Candidate Validation

The operation type is not the only factor limiting the usefulness of a possible surrogate terminal. In many cases, a sprite cannot replace the final geometry when viewed from certain camera positions. For example, Figure 3.4 shows a balcony's final geometry that sticks so far out of the input shape that parts are not contained in the screen space area covered by the input shape. For some surrogate candidates, it is entirely impossible to create fitting surrogate textures. Some candidates are hardly helpful due to a small region of valid camera positions. Other candidates remain valid for a vast range of camera positions even though some vertices in the final geometry protrude significantly from the input shape. Therefore, we suggest a validation process that determines the range of valid camera positions for surrogate candidates and drops unfit candidates.

We consider a camera position C legal relative to a surrogate candidate if all vertices in the final geometry of that candidate can be *mapped* from C onto the candidate's input shape. We say a vertex V can be mapped onto a shape S from a position P or direction D if the line through V and P, or the line through V along D intersects with S. Camera C_1 in Figure 3.5d is an example of an illegal camera position relative to the candidate since V cannot be mapped onto the input shape. On the other hand, camera positions



(a) (b)

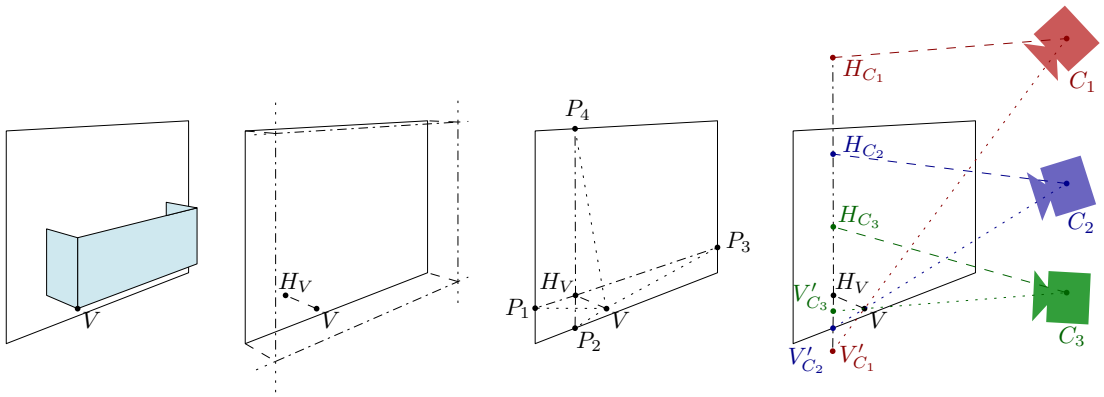
Figure 3.4: Some of the balcony's vertices extend so far out of the facade that they leave the sprite's bounds when viewed from this angle. (a) Input shape highlighted in cyan and final geometry that sticks out too far highlighted in red. (b) Incomplete sprite with cut-off geometry.

C_2 and C_3 are legal since the lines through C_2 and V and C_3 and V intersect the input shape at points V'_{C_2} and V'_{C_3}, respectively. The legal region (LR) of a surrogate candidate is a subset of $\mathbb{R}^3$ and consists of all legal camera positions relative to the candidate. The approximate legal region (ALR) is our approximation of LR. We deem a surrogate candidate to be *fitting* if the ALR is big enough so that it makes sense to check for legality in grammar derivation and dynamically decide whether to insert a sprite.

Algorithm 3.1 analyzes a surrogate candidate and returns a scaling factor κ (see Figure 3.6). κ is used to determine if a camera position is considered legal relative to a hypothetical sprite inserted in place of the surrogate candidate. A camera position with POV indicator $\mathcal{P}$ is considered legal if $|\mathcal{P}_x| < \kappa_x$ and $|\mathcal{P}_y| < \kappa_y$. In case $\kappa = (0, 0)$, there are no or too few legal camera positions for the candidate, so we deem it unfit.

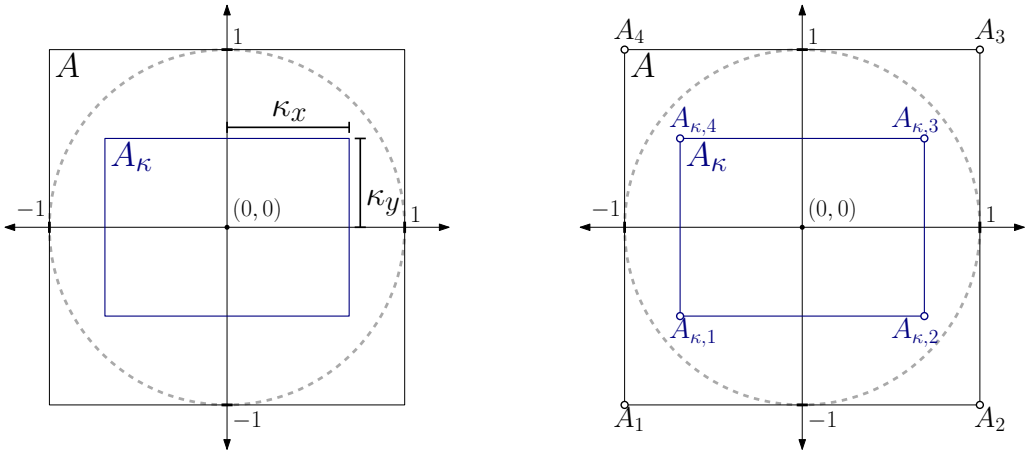
The algorithm iterates over all vertices in the surrogate candidate's final geometry and calculates a preliminary scaling factor κ_V for each vertex V. A first step (lines 6-8) checks whether V can be mapped onto the input shape along the input shape's normal vector (see Figure 3.5b). If mapping the vertex is impossible, the procedure is interrupted, and the surrogate candidate is deemed unfit. While this circumstance does not automatically infer that no legal camera positions exist, it is still reasonable to assume a small LR.

A second step (lines 10-16) finds the closest points P_i to V on the input shape's edges (see Figure 3.5c). The LR for a surrogate candidate considering V is defined by the four planes through each of the candidate edges and V. Each plane divides space into two half-spaces. The intersection of all closed half-spaces that do not contain the input shape is the LR for V. We approximate this vertex-specific LR with a region defined by the two-dimensional vector κ_V. κ_V itself is defined by the minimum of h_1 and h_2 as well as v_1 and v_2 (see line 16), which are the ratio between the lengths of $\overrightarrow{P_iH_V}$ and $\overrightarrow{P_iV}$ (see lines 11-14). Suppose



(a) (b) (c) (d)

Figure 3.5: (a) Example of a surrogate candidate together with its final geometry. A single vertex V is highlighted. (b) A first evaluation step checks if V can be mapped onto the candidate's input shape along the input shape's normal vector. (c) In a second step, the closest point P_i to V on each edge is determined. Vectors $\overrightarrow{P_iV}$ and $\overrightarrow{P_iH_V}$ are then used to calculate a scaling factor κ_V that approximates the range of legal camera positions (see Algorithm 3.1). (d) Example of three camera positions. Contrary to camera positions C_2 and C_3, camera position C_1 is not legal since V cannot be mapped onto the candidate's input shape from C_1.



(a) (b)

Figure 3.6: (a) The scaling factor κ is used to construct a rectangular area A_κ in POV indicator space. All camera positions with $\mathcal{P} \in A_\kappa$ are considered legal by our approach. The circle contains all possible POV indicator values. Area A is the best-case scenario since all camera positions are considered legal. (b) Vertex positions of A are $A_1 = (-1, -1)$, $A_2 = (1, -1)$, $A_3 = (1, 1)$, and $A_4 = (-1, 1)$. A_κ is defined by vertices $A_{\kappa,i} = A_i \circ \kappa$.

Algorithm 3.1: Surrogate candidate fitness determination

Input: surrogate candidate C

1 S = input shape of C in object space
2 S_n = normal vector of S
3 $\kappa = (1, 1)$
4 **foreach** *object space vertex* V *in final geometry of* C **do**
5
6 obtain H_V by mapping V along S_n onto S // see Figure 3.5b
7 **if** H_V *is outside the* S **then**
8 **return** *(0,0)* // C is deemed unfit
9
10 find closest points P_1, P_2, P_3, P_4 to V on edges of S // see Figure 3.5c
11 $h_1 = \|\overrightarrow{P_1H_V}\|/\|\overrightarrow{P_1V}\|$
12 $h_2 = \|\overrightarrow{P_3H_V}\|/\|\overrightarrow{P_3V}\|$
13 $v_1 = \|\overrightarrow{P_2H_V}\|/\|\overrightarrow{P_2V}\|$
14 $v_2 = \|\overrightarrow{P_4H_V}\|/\|\overrightarrow{P_4V}\|$
15
16 $\kappa_V = (min(h_1, h_2), min(v_1, v_2))$ // see Figure 3.7a
17 $\kappa = (min(\kappa_x, \kappa_{V_x}), min(\kappa_y, \kappa_{V_y}))$ // see Figure 3.7b
18 **if** $\kappa_x < \tau_S$ *or* $\kappa_y < \tau_S$ **then** // see Figure 3.7c
19 **return** *(0,0)* // C is deemed unfit
20
21 **return** κ

camera position C has POV indicator $\mathcal{P}$. In case $|\mathcal{P}_x| < \kappa_{V_x} \wedge |\mathcal{P}_y| < \kappa_{V_y}$, we assume that the camera position is legal. Figure 3.7a demonstrates how κ_V can be interpreted visually. This approximation introduces two errors. The first error is that κ_V is calculated relative to vertex V, while $\mathcal{P}$ is calculated relative to the input shape origin. This error introduces both false positive and false negative categorizations and is detrimental for close camera positions but diminishes in significance with increasing camera distance. The second error introduces only false negatives and exists due to the fact that κ_V is a conservative estimate of LR and selects the smaller of the two ratios for each axis (see line 16).

A third step merges κ_V into the overall scaling factor κ (see line 17) and checks whether the new κ still defines a large enough ALR (see line 18). If the ALR is too small, we deem the candidate unfit, as the computational overhead of checking for camera position legality would likely overshadow the performance gain. For this, we compare the individual components of κ to a minimum scaling factor τ_S. If at least one component is lower than τ_S, we return $(0, 0)$ and deem the candidate unfit for surrogate terminal insertion (see Figure 3.7c).

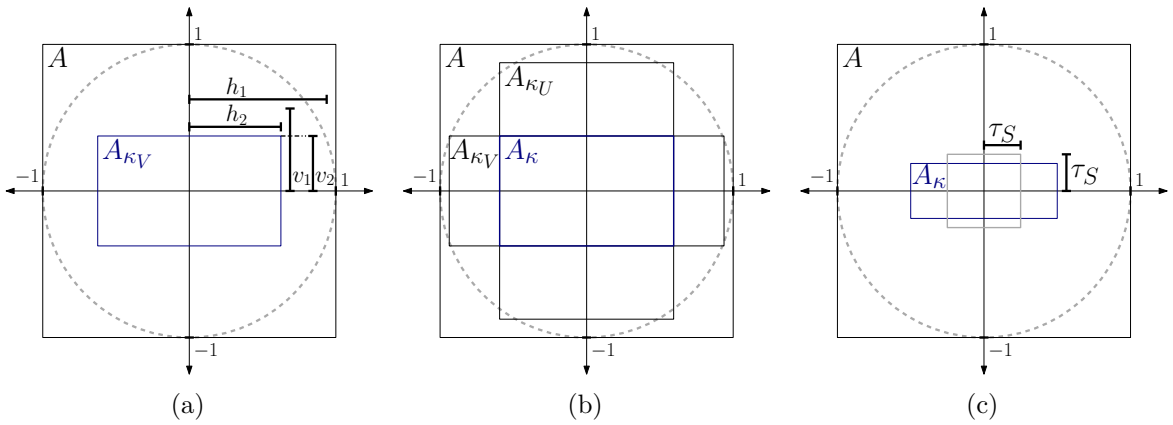


Figure 3.7: (a) Example of a preliminary scaling factor κ_V approximating the LR of a single vertex V. κ_V is determined by scalars h_1, h_2, v_1, and v_2 (see Algorithm 3.1 lines 11-16). (b) We use the component-wise minimum to compute the overall scaling factor κ from all vertex-specific scaling factors (see Algorithm 3.1 line 17). This simplified example calculates the overall scaling factor for the two vertices U and V, with scaling factors κ_U and κ_V. (c) Example of a surrogate candidate deemed unfit due to a κ value with y-component $< \tau_S$.

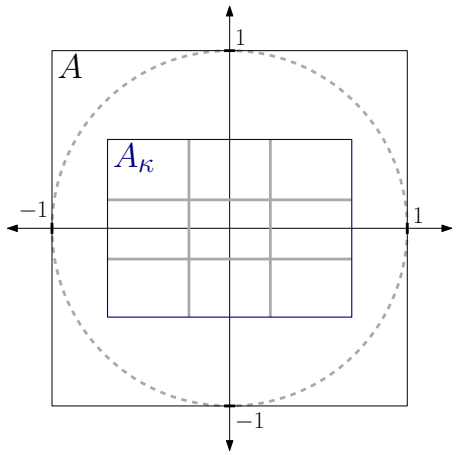


Figure 3.8: Subdivision of the ALR in POV indicator space. Area A_κ is split into a 3×3 grid. Each cell is assigned its own surrogate texture.

3.4 Texture Generation

To generate surrogate textures for a candidate, we first subdivide the ALR in POV indicator space into an $n \times n$ grid of cells (typically $3 \leq n \leq 9$), from now on referenced as the surrogate grid (see Figure 3.8). Then, for each cell, we calculate the center position G, for which we determine a camera position C_G in object space with distance d and unit-length candidate normal n (see Equation 3.2).

$$C_G = d * (\overrightarrow{G} + n * \sqrt{|\overrightarrow{G} \cdot \overrightarrow{G} - 1|}) \tag{3.2}$$

We render the surrogate candidate's final geometry to a framebuffer object (FBO). The exact screen space positions of the surrogate texture within the FBO are found by applying the camera transformation and projection matrices to the vertices of the candidate's input shape. Next, we correct the perspective distortion of the surrogate texture using the FBO as a texture for a quad perpendicular to the camera. The screen space positions of the candidate's input shape serve as UV coordinates of the quad and rectify the image once rendered. Finally, the rectified surrogate texture is added to a global texture atlas containing all surrogate textures for the entire grammar.

3.5 Surrogate Terminal Insertion

A reasonable distance threshold τ_D has to be computed before surrogate terminal operations can be inserted into the AST. Here we adapt the methodology proposed by Steinberger et al. [SKK+14b]. We render both final geometry and sprite from a multitude of camera configurations, making sure to view the geometry from a broad range of directions and distances. The perceived difference between the renders is assessed by an objective Image Quality Assessment (IQA) algorithm (in our case, butteraugli [A+]). We then fit a sigmoid function model to the results using non-linear least squares regression. Equation 3.3 is the sigmoid function we use in our model, and Equation 3.4 is the model itself. Symbols a, b, and c are unknown parameters estimated by the regression technique, and argument x in Equation 3.4 is the camera distance. Once values a, b, and c have been approximated, we intersect the function with a predetermined acceptable error threshold τ_E to get the distance threshold τ_D (see Figure 3.9).

$$s(x) = \frac{1}{1 + e^{-x}} \tag{3.3}$$

$$err(x) = s(a * (x + b)) * c \tag{3.4}$$

Now that τ_D is calculated, surrogate terminal operations are inserted into the AST in front of fit surrogate candidates. τ_D and κ are added as parameters of the terminal operation, together with positional information of the surrogate textures within the texture atlas. Finally, the modified AST is translated back into conventional shape grammar notation, that other tools, which implement the proposed surrogate terminal operation, may use to generate 3D models more efficiently. The code blocks in Figure 3.10 show the example grammar from Figure 3.2 together with a more elaborate pseudo-code version that demonstrates how surrogate terminals work internally.

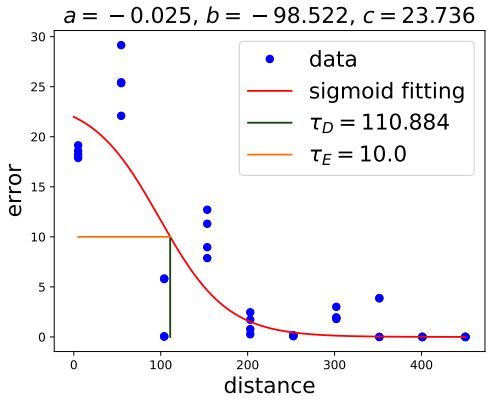


Figure 3.9: τ_D gets approximated by fitting a sigmoid function model to the visual error data and intersecting it with a predetermined error threshold τ_E.

3.6 Frustum Pruning Operation

To further increase the efficiency of our method, we decided to add a frustum pruning operation. It checks whether the final geometry produced by the operation's child rules is visible given the current camera configuration and stops derivation if it is not. For each surrogate candidate (regardless of fitness), we find the most distant vertex V to the input shape's origin O. We use the distance between O and V as the radius for an approximate bounding sphere, which the frustum pruning operation uses when executed. We insert the operation directly before surrogate candidates regardless of fitness (see Figure 3.11) since frustum pruning is applicable even if it is impossible to create a valid impostor.

```
Start:
  Extrude(3)
  CompSplit() {
    mantle:
      Repeat(+X, 5)
      Facade;
    default: Generate;
  };

Facade:
  SubDiv(+X) {
    1.0 : Decoration;
    1.0r: Wall;
    2.5 : Glass;
    1.0r: Wall;
    1.0 : Decoration;
  };

Wall:
    ...

...
```

(a)

```
Start:
  Extrude(3)
  CompSplit() {
    mantle:
      Terminal(κ¹, τ¹_D, TLoc¹)
      Repeat(+X, 5)
      Facade;
    default: Generate;
  };

Facade:
  Terminal(κ², τ²_D, TLoc²)
  SubDiv(+X) {
    1.0 : Decoration;
    1.0r: Wall;
    2.5 : Glass;
    1.0r: Wall;
    1.0 : Decoration;
  };

Wall:
    ...

...
```

(b)

```
Start:
  Extrude(3)
  CompSplit() {
    mantle:
      If(dist() ≥ τ¹_D and P in κ¹) {
        id = getTex(P, κ¹, n)
        Sprite(TLoc¹, id);
      }Else {
        Repeat(+X, 5)
        Facade;
      };
    default: Generate;
  };

Facade:
  If(dist() ≥ τ²_D and P in κ²) {
    id = getTex(P, κ², n)
    Sprite(TLoc², id);
  }Else {
    SubDiv(+X) {
      1.0 : Decoration;
      1.0r: Wall;
      2.5 : Glass;
      1.0r: Wall;
      1.0 : Decoration;
    };
  };

Wall:
    ...

...
```

(c)

Figure 3.10: A thorough example of a shape grammar modified by the surrogate terminal operations. We use the example grammar shown in Figure 3.2. (a) The original shape grammar. Possible surrogate candidate rules are highlighted. (b) The same grammar after being modified. κ^i, τ_D^i, and $TLoc^i$ are surrogate terminal–specific constants. $TLoc^i$ specifies the texture atlas positions of all surrogate textures of the surrogate terminal in question. (c) Pseudo-code explanation of what the surrogate terminal operation does internally. $dist()$ returns the distance between the camera and the input shape's origin. $\mathcal{P}$ is the POV indicator of the camera position relative to the input shape. The global constant n defines the resolution of the surrogate grid. $getTex(...)$ determines which surrogate texture should be used based on $\mathcal{P}$, κ^i, and the surrogate grid resolution. Finally, $Sprite(...)$ replaces the input shape with a quad of equal dimensions and sets the correct material properties.

```
Start:
  Extrude(3)
  CompSplit() {
    mantle:
      Repeat(+X, 5)
      Facade;
    default: Generate;
  };

Facade:
  SubDiv(+X) {
    1.0 : Decoration;
    1.0r: Wall;
    2.5 : Glass;
    1.0r: Wall;
    1.0 : Decoration;
  };

Wall:
    ...

...
```

(a)

```
Start:
  FrustumCheck(rad^1)
  Extrude(3)
  CompSplit() {
    mantle:
      FrustumCheck(rad^2)
      Terminal(κ^1, τ_D^1, TLoc^1)
      Repeat(+X, 5)
      Facade;
    default: Generate;
  };

Facade:
  FrustumCheck(rad^3)
  Terminal(κ^2, τ_D^2, TLoc^2)
  SubDiv(+X) {
    1.0 : Decoration;
    1.0r: Wall;
    2.5 : Glass;
    1.0r: Wall;
    1.0 : Decoration;
  };

Wall:
    ...

...
```

(b)

```
Start:
  If(!inFrust(rad^1)) {
    Discard;
  }Else {
    Extrude(3)
    CompSplit() {
      mantle:
        FrustumCheck(rad^2)
        Terminal(κ^1, τ_D^1, TLoc^1)
        Repeat(+X, 5)
        Facade;
      default: Generate;
    };
  };

Facade:
  If(!inFrust(rad^3)) {
    Discard;
  }Else {
    Terminal(κ^2, τ_D^2, TLoc^2)
    SubDiv(+X) {
      1.0 : Decoration;
      1.0r: Wall;
      2.5 : Glass;
      1.0r: Wall;
      1.0 : Decoration;
    };
  };

Wall:
    ...

...
```

(c)

Figure 3.11: A thorough example of a shape grammar modified by surrogate terminal operations and frustum pruning operations. We use the example grammar shown in Figure 3.2. (a) The original shape grammar. Possible surrogate candidate rules are highlighted. (b) The same grammar after being modified. Frustum pruning operations are highlighted in blue, while surrogate terminal operations are highlighted in green. rad^i is an operation-specific constant that defines the radius of a bounding sphere. (c) Pseudo-code explanation of what the frustum pruning operation does internally. The second frustum pruning operation has not been expanded to increase clarity. $inFrust(...)$ returns true if the bounding sphere positioned at the origin of the input shape with radius rad^i is inside the current view frustum.

CHAPTER 4

Evaluation & Comparison

To evaluate our solution, we set up three different shape grammars that generate scenes of varying scale and complexity (original grammar). On each original grammar, we apply our method to get a modified grammar. We also generate a separate shape grammar that uses view-independent surrogate terminals as suggested by Steinberger et al. [SKK+14b] (steinberger grammar) to compare our results to previous work.

The original *dorms* grammar generates the smallest scene with approximately $1.7M$ vertices. It consists of 16 buildings with high detail, each house using roughly $100K$ vertices. The buildings feature inset windows and balconies, which are repeated over four to nine floors.

The original *town* grammar combines multiple building archetypes in one grammar and arranges them in a street grid of 10-by-10 blocks. It generates $12.2M$ vertices and is the most complex grammar, as it uses the most rules to create the different buildings, and the most surrogate terminals have to be inserted.

The original *balcony* grammar generates the biggest scene with around $335.7M$ vertices. Here, $10K$ high-rise buildings with balconies that stick out of each floor make up the scene. The individual buildings themselves are not that highly detailed, use about $33.6K$ vertices each, and have 11 to 19 floors.

To generate the modified grammar, we used a 5-by-5 surrogate grid, a minimum scaling factor $\tau_S = 0.1$, and an acceptable error threshold $\tau_E = 20$. The acceptable error threshold used to generate the steinberger grammars was also chosen to be 20.

4.1 Qualitative Analysis

Table 4.1 shows representative example renders generated from all three scenarios with the above-mentioned three distinct grammar versions. In contrast to our method, Steinberger's approach inserts surrogate terminals even if some vertices of the approximated

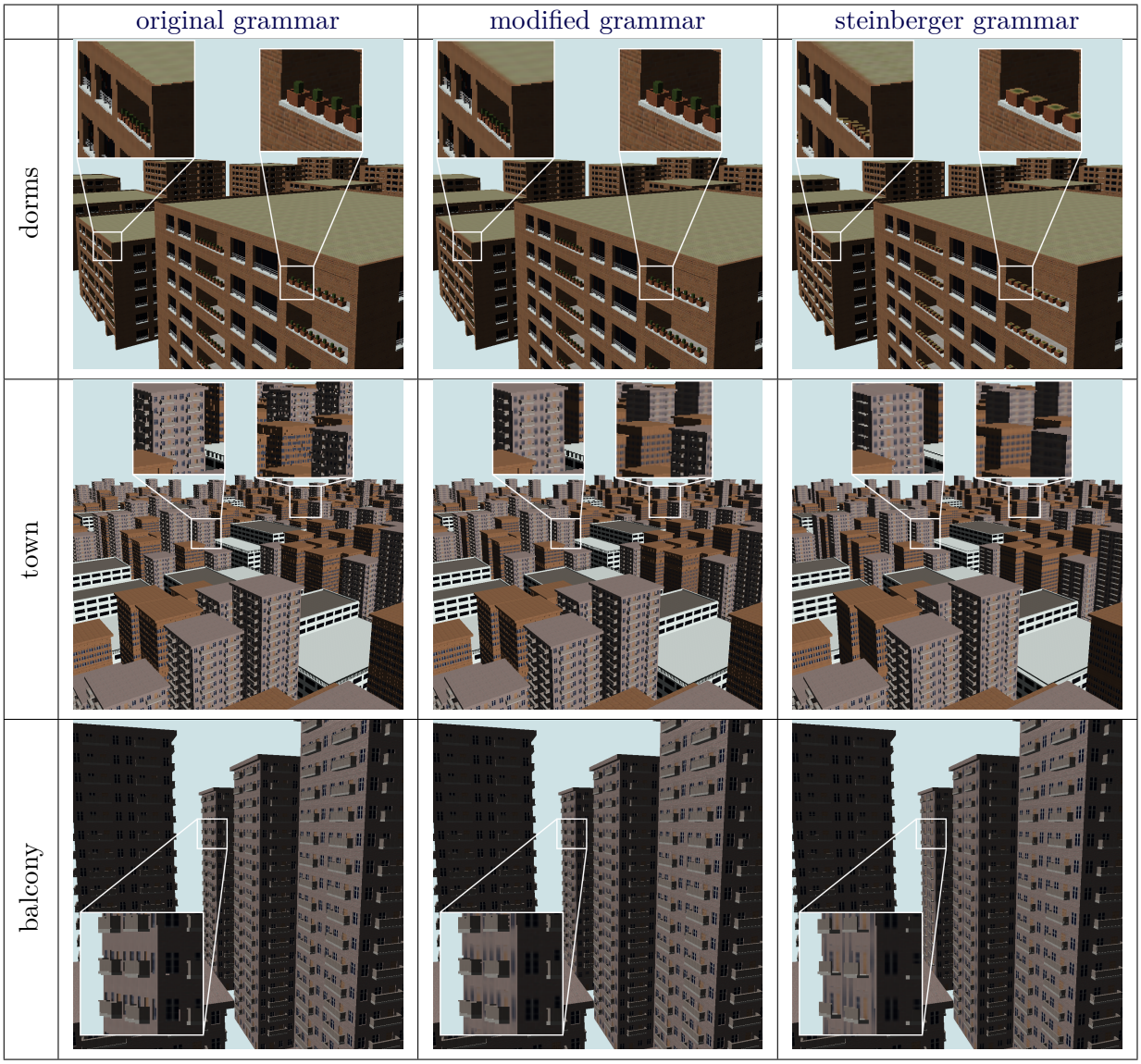


Table 4.1: Example renders for each of the three test scenarios using the original grammar, modified grammar, and steinberger grammar, respectively.

geometry cannot be mapped onto the input shape. Consequently, steinberger grammars sometimes generate visually incomplete geometry, such as balconies without railings and flat flowerpots (see balcony- and dorms grammar in Table 4.1). Our method avoids these issues while still being able to reduce a significant amount of geometric detail.

Scenes generated by original grammars and modified grammars look similar. However, the less detailed geometry derived by the original grammar is subject to fewer aliasing issues previously caused by far-away triangles of sub-pixel size (see town grammar in Table 4.1).

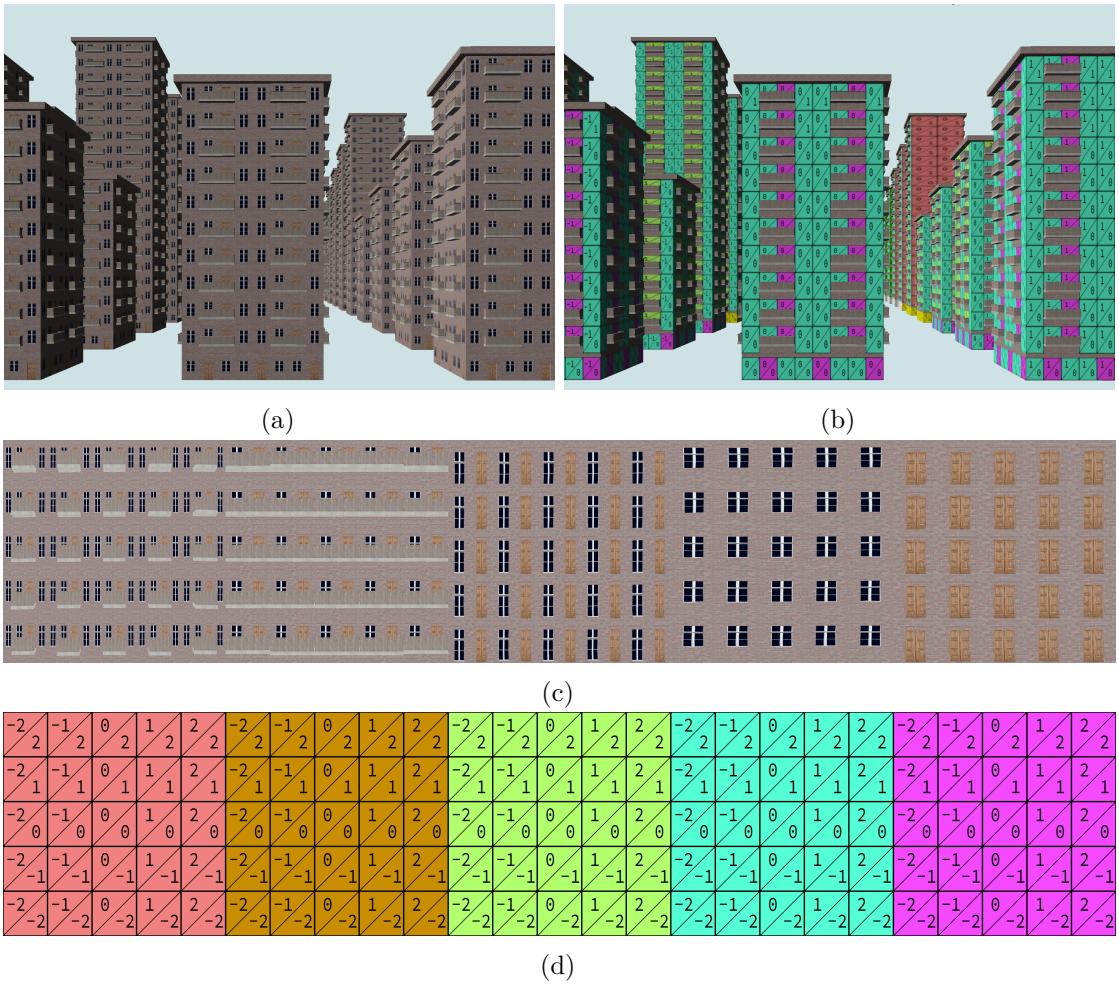


Figure 4.1: (a) Example render of geometry produced by the modified grammar of the balcony scenario. (b) The same render with replaced surrogate textures. Each surrogate terminal operation is assigned a unique color. Surrogate terminals that approximate the geometry of the balconies (red) use an unreasonably high distance threshold τ_D. (c) Extract of the texture atlas used for rendering (a). (d) The same extract of the modified texture atlas as (c) used for rendering (b). Note that the orange surrogate textures are not used once in (b).

Contrary to our expectations, the modified grammars seemed to mainly use surrogate terminals of flat geometry. More complex geometry with features like balconies extending into or out of the surrogate's input shape only got replaced by sprites in specific cases (see Figure 4.1). We attribute this to a too strict calculation of distance threshold τ_D and scaling factor κ, leading to an unnecessarily deep derivation of the grammar.

	vertex count			derivation time			rendering time			visual diff.	
	umod	ours	stb	umod	ours	stb	umod	ours	stb	ours	stb
dorms	1.7 M	947.3 K	183.6 K	1.0s	1.8s	0.6s	1.2ms	0.9ms	0.6ms	31.7	47.2
town	12.2 M	4.0 M	852.2 K	1.9s	2.6s	1.0s	6.6ms	2.8ms	1.3ms	23.9	58.3
balcony	335.7 M	45.0 M	3.5 M	120.4s	4.2s	0.7s	10.0s	18.2ms	2.7ms	46.5	56.4

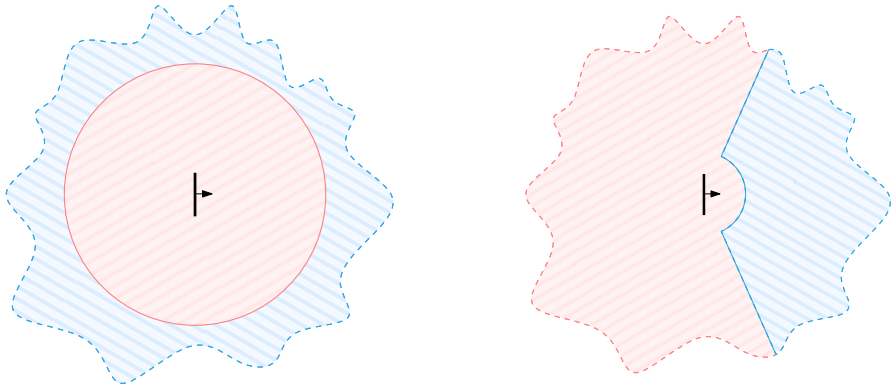
Table 4.2: Average test results for each scenario using grammars *without* frustum pruning. The unmodified original grammars are abbreviated as *umod*, the modified grammars generated by our method are shortened to *ours*, and steinberger grammars are referred to as *stb*.

Another issue we identified is that some surrogate terminals rarely get used since their region of adequate camera positions (adequate region (AR)) is roughly contained within the AR of surrogate terminals higher up in derivation hierarchy (*parents*). Therefore, in case a parent surrogate terminal cannot terminate grammar derivation, there is a low chance the surrogate terminal itself can be applied. Figure 4.1d demonstrates this issue since the orange surrogate texture cannot be found within Figure 4.1b. The surrogate terminal with red textures contains the orange surrogate terminal and uses a scaling factor $\kappa = (0.93, 0.72)$ and distance threshold $\tau_D = 213.5$, while the orange terminal uses $\kappa = (0.13, 0.72)$ and $\tau_D = 130.9$. Since the orange terminal gets used in very few scenarios but has to be derived every time the red surrogate terminal cannot stop grammar derivation, its computational overhead likely overshadows the benefit it provides.

4.2 Quantitative Analysis

For each scenario, we picked at least 100 camera configurations showing the buildings from different viewing directions and -distances. We specifically focused on geometry produced by rules either method could have considered as surrogate candidate. We derived the geometry from the grammars using these camera configurations and compared them based on vertex count, derivation time, render time, and visual difference. Vertex count is the number of generated vertices by the grammar. Derivation time states how long it takes to generate geometry by deriving the grammar in question. Rendering time refers to the time it takes to render the already generated geometry. Finally, the visual difference is how closely the images rendered by the altered grammars match the originals based on the Image Quality Assessment (IQA) tool butteraugli [A$^+$]. We used one machine equipped with an Intel Core i7-8700K CPU 3.70 GHz, an NVIDIA GeForce GTX 1080 Ti, and 32GB RAM for all of our tests.

In a first test, we used surrogate terminal operations alone without the aid of the frustum pruning operation to see how they perform in isolation. Table 4.2 shows that both the modified grammars and steinberger grammars achieve enormous improvements over the original grammars. Especially vertex count and render time could be reduced drastically, Steinberger's method generally performing better than our method. In small- and



(a) (b)

Figure 4.2: 2D representations of space partitions into adequate region (AR) (blue) and inadequate region (IAR) (red) by a surrogate terminal (black). (a) Space partition by Steinberger's method [SKK+14b]. (b) Space partition by our method. Note that the IAR in (a) is contained in a closed surface and therefore has a finite volume, while it has infinite size in (b).

medium-sized scenes, the additional overhead of our method during grammar derivation outweighs the added benefit. With growing scene size, however, it becomes more and more infeasible to derive the whole grammar. For the *balcony* scenario, our solution required 3.5% of the derivation time that the original grammar needed.

Compared to Steinberger's grammars, our modified grammars tend to generate more visually correct images, but they take more time and produce more vertices. We attribute this to two factors:

1. The insertion criteria for surrogate terminals are harsher since we avoid inserting invalid sprites.
2. Checking insertion criteria is more resource-intensive since we use both viewing direction and camera distance.

Figure 4.2 shows a 2D representation of the AR of surrogate terminals by Steinberger's method and our method. While the region of inadequate camera positions (IAR) stays finite in Steinberger's method, it is infinitely big in our solution. Consequently, our method can insert surrogate terminals less frequently and has to process the derivation tree more thoroughly.

		vertex count			derivation time			render time			visual diff.	
		umod	ours	stb	umod	ours	stb	umod	ours	stb	ours	stb
dorms	surr	1.7 M	947.3 K	183.6 K	1.0s	1.8s	0.6s	1.2ms	0.9ms	0.6ms	31.7	47.2
	+ prune		177.6 K	33.1 K		0.9s	0.5s		0.6ms	0.5ms		
	% gain	-	81.3%	82.0%	-	52.7%	10.9%	-	33.0%	24.5%	-	-
town	surr	12.2 M	4.0 M	852.2 K	1.9s	2.6s	1.0s	6.6ms	2.8ms	1.3ms	23.9	58.3
	+ prune		690.9 K	117.2 K		2.1s	0.8s		0.9ms	0.8ms		
	% gain	-	82.6%	86.2%	-	17.1%	15.2%	-	66.8%	35.8%	-	-
balcony	surr	335.7 M	45.0 M	3.5 M	120.4s	4.2s	0.7s	10.0s	18.2ms	2.7ms	46.5	56.4
	+ prune		8.0 M	418.9 K		1.2s	0.4s		4.2ms	0.9ms		
	% gain	-	82.2%	88.2%	-	71.8%	39.3%	-	77.0%	67.2%	-	-

Table 4.3: Average test results for each scenario where grammars use surrogate terminals alone (surr) or surrogate terminals together with frustum pruning operations (+ prune). An extra row (% gain) shows the performance gain achieved by adding the frustum pruning operation. The unmodified original grammars are abbreviated as *umod*, the modified grammars generated by our method are shortened to *ours*, and steinberger grammars are referred to as *stb*.

To limit the cases in which our method falls back on generating the entire geometry for off-screen elements, we added the frustum pruning operation described in Section 3.6. Table 4.3 shows the test results for all shape grammars with and without inserted frustum pruning operations. An extra row further highlights the performance gain achieved by adding the new operations. Even though the improvement was not enough for our method to surpass Steinberger's method, performance increases on derivation time had a more substantial effect on our method across the board. This suggests that more advanced visibility pruning approaches such as stricter frustum pruning or occlusion pruning can help mitigate the difference between the two techniques.

CHAPTER 5

Discussion & Future Work

In this chapter, we discuss how various shortcomings of our solution can be addressed in future work. At first, further optimization opportunities of the visibility pruning process are described. We then revisit our way of determining whether a surrogate terminal should insert a sprite and point out multiple ways to improve upon it. Lastly, additional criteria are highlighted that can help identify redundant surrogate candidates.

5.1 Visibility Pruning

As stated in Section 4.2, future work could further increase performance by adding or improving visibility pruning methods. The frustum pruning approach we use tries to avoid unnecessary overhead and uses bounding spheres that are quick to check against the frustum planes. Consequently, it is lenient and may prune fewer elements than a more sophisticated technique. Our frustum pruning operation can be improved in two ways: The inaccurate bounding spheres could be replaced by tighter bounding volumes such as oriented bounding boxes. The operation could also check whether a bounding volume is entirely within the viewing frustum. If this is the case, the derivation cannot be stopped. However, child operations no longer need to check for frustum pruning since they are contained in the parents bounding volume and are therefore also contained within the view frustum.

5.2 Adequate Camera Regions

Visibility pruning alone might not provide enough improvement. Inserting frustum pruning operations in steinberger grammars also increases performance. Since Steinberger's method uses a less complicated decision-making process to determine whether to use a surrogate terminal, we can generally assume that derivation of a single surrogate terminal takes longer using our method. Unless our method can terminate derivation earlier than

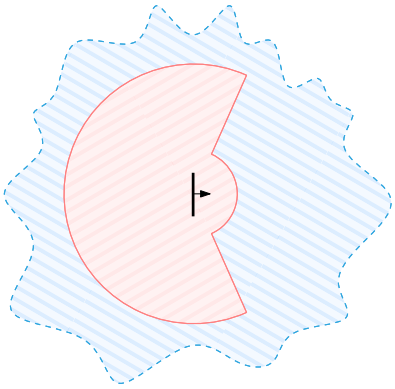
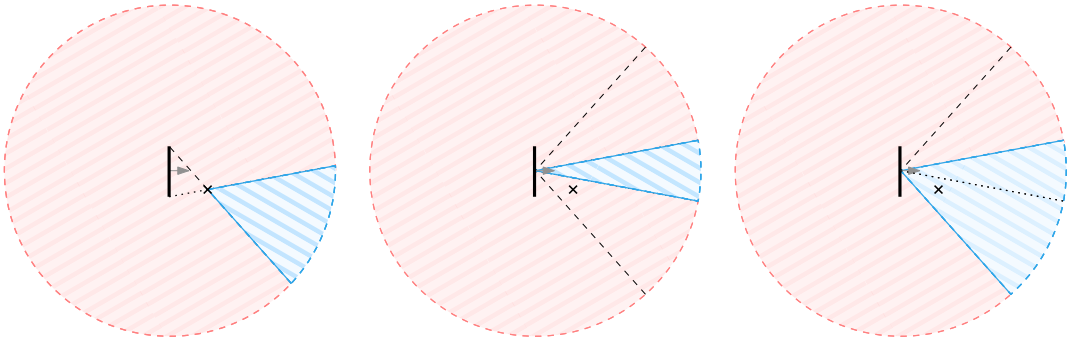


Figure 5.1: The introduction of a second distance threshold restricts inadequate camera positions to a finite volume. This trade-off reintroduces the generation of incomplete geometry at great distances.

Steinberger's method, it is unlikely that we reduce derivation time to a point where our method is generally faster than Steinberger's approach. Figure 4.2 suggests that our proposed surrogate terminal can terminate grammar derivation in fewer cases. This is because the inadequate region (IAR) for our surrogate terminals is infinitely big, as opposed to Steinberger's surrogate terminals, where the IAR has a finite size. Future work could look at multiple ways to improve on this front.

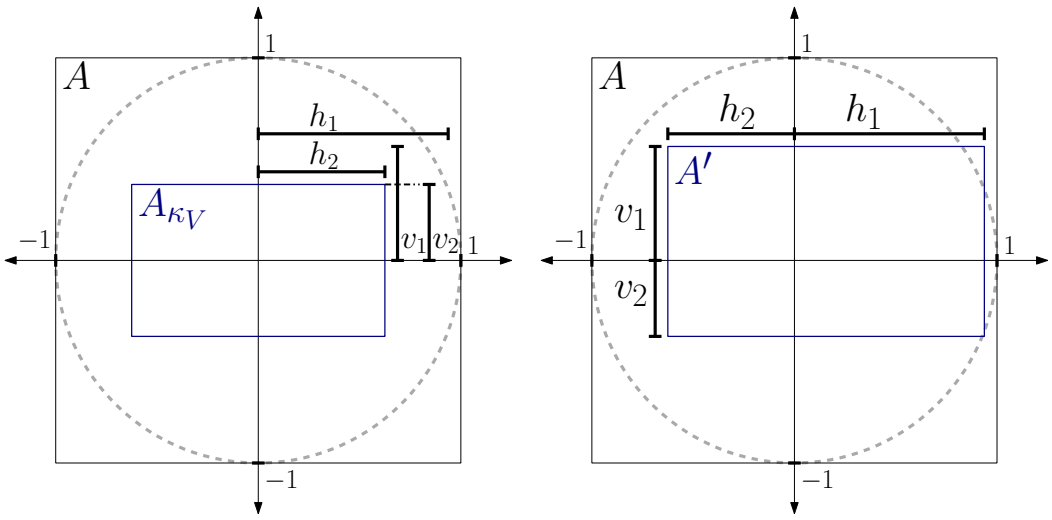
One possible enhancement would be to find a second distance threshold, beyond which the Human Visual System (HVS) can no longer perceive the geometric detail added by further deriving the grammar regardless of viewing direction. Generating incomplete geometry at such distance would go unnoticed, therefore inserting invalid sprites can reduce vertex count, derivation time, and render time without affecting visual quality. This change essentially combines our approach with that of Steinberger et al. [SKK+14b]. It limits the IAR of surrogate terminals to a volume of finite size (see Figure 5.1).

Another improvement could be made by calculating an additional offset vector $\overrightarrow{w}$ for the surrogate grid. Just scaling the grid by vector κ underestimates the legal region (LR) conservatively. Using κ alone, the grid always remains centered on the origin. Suppose a camera position with POV indicator (x, y) is illegal relative to the surrogate candidate. In that case, camera positions with POV indicators $(-x, y)$, $(x, -y)$, and $(-x, -y)$ are also automatically deemed illegal even though they may not be. Shifting the surrogate grid by some vector $\overrightarrow{w}$ removes this constraint and can lead to larger, more accurate approximate legal regions (ALRs). Figure 5.2 shows how this change could modify space partitioning for a single vertex. Figure 5.3 shows the impact this change can have on the surrogate grid.



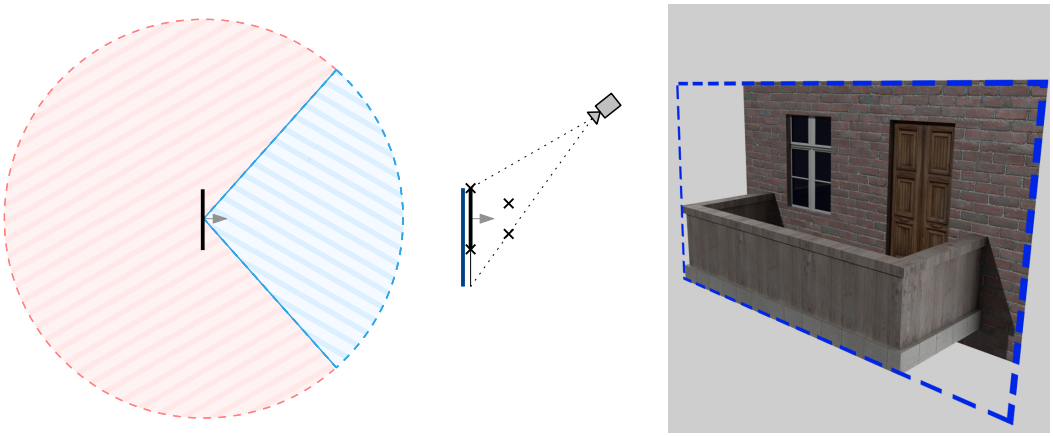
(a) (b) (c)

Figure 5.2: (a) Correct space partitioning into legal region (LR) (blue) and illegal region (red) for a single vertex (×) relative to a surrogate candidate (bold line). (b) Our method approximates the space partitioning in (a) using only scaling factor κ. Depending on vertex position, this partition in approximate legal region (ALR) (blue) and approximate illegal region (red) can be quite inaccurate and stricter than it needs to be. (c) A modification to our approach might use an offset vector $\overrightarrow{w}$ in addition to κ to better approximate the correct space partitioning in (a).



(a) (b)

Figure 5.3: (a) Our method considers any camera position with POV indicator in area A_{κ_V} adequate for sprite insertion. We compute κ_V by finding scalars v_1, v_2, h_1, and h_2 (see Section 3.3) and then assigning $\kappa_{V_x} = min(v_1, v_2)$ and $\kappa_{V_y} = min(h_1, h_2)$. This conservative estimate wrongly classifies many legal camera positions as illegal. (b) The introduction of an additional offset vector $\overrightarrow{w_V}$ that shifts area A' by a fixed amount can help better approximate the LR. κ_V is now computed as $\kappa_V = (h_1 + h_2, v_1 + v_2)/2$, while $\overrightarrow{w_V} = (h_1 - h_2, v_1 - v_2)/2$.



(a) (b) (c)

Figure 5.4: (a) Per surrogate candidate (bold line), a desired LR decided upon, for which each generated surrogate texture should be valid (blue area). (b) Upon surrogate texture generation, the candidate's final geometry (vertices marked by ×) is used to determine each sprite's (blue line) required size and position relative to the input shape. Since this calculation is view-dependent, the values for position and size change for each sprite of a surrogate candidate. (c) Example of a sprite (blue outline) that is bigger than its input shape.

One final idea for augmentation would be to make sprites bigger than their respective input shape. This way, geometry sticking out of the input shape's screen space area could still be replaced by a valid impostor (see Figure 5.4). Surrogate terminals would need to implement nailboard behavior (see Section 2.2) to avert visibility issues with adjacent geometry.

5.3 Unfit Candidate Detection

Surrogate candidate validation determines if a candidate by itself has a sufficiently large ALR. However, as noted previously in Section 4.1, it currently goes unchecked if other surrogate candidates higher up in the derivation hierarchy already cover the same region. Under certain circumstances, *weak* surrogate terminals might be inserted into a shape grammar. These weak surrogate terminals are either unable to terminate grammar derivation or only terminate for a narrow range of camera positions. Therefore, the benefits of weak surrogate terminals are overshadowed by the computational overhead, and marking them as unfit increases overall performance.

Future work could compare the adequate region (AR) for surrogate candidates with the already covered region by previous surrogate terminals to help decide if a candidate should be deemed fit. An even broader approach could achieve a similar effect by finding a way to estimate the expected performance change that comes with inserting a surrogate

terminal into grammar and deciding whether to insert a given surrogate terminal based on this estimation.

CHAPTER 6

Conclusion

We have presented a novel method to insert view-dependent surrogate terminal (VDST) operations into a shape grammar. Shape grammars utilizing VDSTs produce significantly less complex geometry. Generated geometry consequently requires less time to render and suffers from fewer aliasing artifacts. In contrast to view independent approaches, VDSTs are able to avoid generating incomplete geometry at a distance. Our approach can approximate geometry that extends a significant distance into or out of the original surface. However, the resulting surrogate terminals are often heavily limited in their range of legal camera positions. Further work is required to make surrogate terminals of protruding geometry viable for broader application. The attained benefits come at the cost of some derivation time. Also, under certain circumstances, more geometry is generated than by previous approaches. However, plenty of room for improvement remains.

The introduction of a rudimentary frustum pruning operation demonstrated the possible performance gain by combining surrogate terminals with visibility pruning techniques. Future work can further increase performance by employing more sophisticated visibility pruning approaches. More accurate space divisions into approximate legal and illegal camera regions can be found. Further methods can be developed that weigh the possible gains of a surrogate candidate against its computational overhead to decide whether inserting it into the grammar provides any advantage.

Overall, further development is necessary to compete on a level with previous work. Nevertheless, our method provides promising results that indicate the concept is applicable in more scenarios than earlier techniques. With further optimization, we expect our method to outperform prior approaches.

Glossary

adequate If a surrogate terminal would terminate grammar derivation based on a given camera position, the camera position is considered adequate relative to that surrogate terminal. View-independent surrogate terminals proposed by Steinberger et al. [SKK+14b] consider a camera position adequate if the distance between camera and input shape origin is greater than a predetermined distance threshold τ_D. View-dependent surrogate terminals (VDSTs) additionally require the camera position to be contained within the approximate legal region (ALR) of the surrogate terminal. 34, 35, 38, 39, 45, 46

adequate region (AR) The set of camera positions considered adequate by a surrogate terminal. 34, 35, 40, 46, 49

affine transformation A set of geometric transformations that include but are not limited to translation, scaling, rotation, and any combination of these three [Ebe01]. 7, 46

approximate legal region (ALR) The approximation of the legal region (LR) by a surrogate candidate or surrogate terminal. 23, 25, 26, 38–40, 45, 49

billboard Camera-aligned sprite. 9, 11, 46

butteraugli Objective Image Quality Assessment (IQA) method that tries to quantify the perceived difference between two images. 14, 27, 34

derivation The process of applying a shape grammar in order to generate geometry. An axiom (tuple of initial shape and starting symbol) is defined and rules of the grammar are applied recursively until no non-terminal shape remains. Not to confuse with evaluation. 2, 6–8, 14, 15, 17, 18, 20–23, 28, 32–38, 40, 43, 45, 46

evaluation The process of analyzing a shape grammar with the intent of modifying it. Not to confuse with derivation. 18, 24, 45

fit Our method deems a surrogate candidate fit, if its ALR is big enough. 22, 23, 25–28, 40

image-based impostor Object within a virtual 3D scene that uses an image to efficiently approximate complex geometry, which would otherwise be less efficient to render in all its detail. 5, 8, 12, 14, 47

inadequate region (IAR) Opposite of the adequate region (AR). The set of camera positions considered inadequate by a surrogate terminal. 35, 38, 49

input shape The shape on which an operation is executed. The input shape gets replaced by the output shape of the operation, once it has finished executing. 17, 18, 20–25, 27–30, 32, 33, 40, 45, 46

legal A camera position is legal relative to a surrogate candidate / surrogate terminal if all vertices within the final geometry of that candidate / terminal can be mapped onto the candidate's / terminal's input shape from the camera position. 22–25, 38, 39, 43, 46

legal region (LR) The set of legal camera positions relative to a surrogate candidate or surrogate terminal. 23, 25, 26, 38–40, 45, 49

modified grammar A modified version of a shape grammar that *does* make use of surrogate terminals. 18, 19, 31–36

nailboard A billboard that stores the z-buffer offset of each pixel relative to the billboard's surface, so that visibility can be caluclated correctly. 9, 40

object space 3D coordinate space of an object. All coordinates are specified relative to the object's origin, which lies at $(0, 0, 0)$. 9, 20, 21, 25, 26

operation Helps defining a shape grammar. Introduced by Müller et al. [MWH+06], an operation is part of a rule. It takes an input shape, modifies it in some way (affine transformation, duplication, conditional selection between multiple child operations, etc.), and replaces the shape with the result of the modification. Multiple operations can be chained together within a rule. 7, 14, 15, 17, 18, 20–22, 27–30, 33, 34, 36, 37, 43, 46, 47

original grammar An unmodified version of a shape grammar that *does not* make use of surrogate terminals. 18, 19, 31, 32, 34–36

per-pixel difference Objective IQA method that compares two images with equal size by comparing each pixel of the first image with its positional counterpart in the second image and averaging the difference. 13, 14

rule A rule of a shape grammar that defines the production guidelines used to derive the described language (the set of all possible shapes). It determines which shape or set of shapes may be replaced by another set of shapes. 5–8, 14, 21, 22, 28–31, 45–47

screen space 2D coordinate space of a rendered image. 22, 27, 40

shape grammar A grammar that operates on 2D or 3D shapes. Proposed initially by George Stiny et al. [SG71] in 1971, they have since been adapted and modified to allow the procedural generation of 3D models. 2, 3, 5–8, 14, 15, 17, 18, 20, 21, 27, 29–31, 40, 43, 45–47

sprite Image-based impostor that displays a texture of the geometry it approximates on a quad. 9–12, 14, 15, 17, 18, 20–23, 27, 33, 35, 37–40, 45, 47

steinberger grammar A shape grammar modified by surrogate terminals as suggested by Steinberger et al. [SKK+14b]. We compare our method to previous work by comparing bench marks between shape grammars modified by our surrogate terminals and Steinbergers surrogate terminals. 31, 32, 34, 36, 37

surrogate candidate Operation inside a shape grammar rule that might be an appropriate insertion point for a surrogate terminal. 18, 22–30, 34, 37–40, 43, 45–47

surrogate grid Subdivided area in POV indicator space that is valid for a given surrogate candidate. Each cell in this grid is associated with its own surrogate texture. 26, 29, 31, 38

surrogate terminal Operation that decides whether subsequent operations should be evaluated or if a sprite with appropriate surrogate texture should be inserted into the scene. 2, 14, 15, 17, 18, 20–22, 25, 27, 29–31, 33–38, 40, 41, 43, 45–47

surrogate texture Texture for a sprite that can be used instead of evaluating subsequent operations inside a shape grammar. 2, 17, 18, 20–22, 26, 27, 29, 33, 34, 40, 47

texture atlas Image that contains textures for multiple faces or objects, packed in a space-efficient way. 27, 29, 33

valid An image-based impostor is valid relative to a camera position, or a camera position is valid relative to an image-based impostor, if the image-based impostor looks *plausible* when rendered from the camera position. The image-based impostor looks plausible, if the visual difference between the geometry the image-based impostor represents and the image-based impostor itself is below a predefined margin of error. 9–11, 17, 18, 22, 28, 35, 38, 40, 47

Acronyms

ALR approximate legal region. 23, 25, 26, 38–40, 45, *Glossary:* approximate legal region (ALR)

AR adequate region. 34, 35, 40, 46, *Glossary:* adequate region (AR)

AST Abstract Syntax Tree. 18, 21, 22, 27

FBO framebuffer object. 27

GPU Graphics Processing Unit. 2, 8, 15

HVS Human Visual System. 17, 38

IAR inadequate region. 35, 38, *Glossary:* inadequate region (IAR)

IQA Image Quality Assessment. 5, 8, 13, 14, 27, 34, 45, 46

LOD level of detail. 2, 8, 9, 11

LR legal region. 23, 25, 26, 38–40, 45, *Glossary:* legal region (LR)

MS-SSIM Multi-Scale Structural Similarity. 14

POV indicator point of view indicator. 20, 21, 23–26, 29, 38, 39, 47

SSIM Structural Similarity. 14

VDST view-dependent surrogate terminal. 2, 3, 43, 45

Bibliography

[A+] Jyrki Alakuijala et al. Butteraugli, 2016. *URL https://github. com/google/butteraugli*.

[AM00] Ulf Assarsson and Tomas Möller. Optimized view frustum culling algorithms for bounding boxes. *J. Graphics, GPU, & Game Tools*, 5(1):9–22, 2000.

[AOS+17] Jyrki Alakuijala, Robert Obryk, Ostap Stoliarchuk, Zoltan Szabadka, Lode Vandevenne, and Jan Wassenberg. Guetzli: Perceptually guided JPEG encoder. *CoRR*, abs/1703.04421, 2017.

[CCSD03] Daniel Cohen-Or, Yiorgos Chrysanthou, Cláudio T. Silva, and Frédo Durand. A survey of visibility for walkthrough applications. *IEEE Trans. Vis. Comput. Graph.*, 9(3):412–431, 2003.

[Chi95] Norman Chin. Iii.5 - a walk through bsp trees. In Alan W. Paeth, editor, *Graphics Gems V*, pages 121–138. Academic Press, Boston, 1995.

[DDSD03] Xavier Décoret, Frédo Durand, François X. Sillion, and Julie Dorsey. Billboard clouds for extreme model simplification. *ACM Trans. Graph.*, 22(3):689–696, 2003.

[Ebe01] David H. Eberly. *3D game engine design - a practical approach to real-time computer graphics*. Morgan Kaufmann, 2001.

[HMVI13] Mark Hendrikx, Sebastiaan A. Meijer, Joeri Van Der Velden, and Alexandru Iosup. Procedural content generation for games: A survey. *ACM Trans. Multim. Comput. Commun. Appl.*, 9(1):1:1–1:22, 2013.

[Ios11] Alexandru Iosup. POGGI: generating puzzle instances for online games on grid infrastructures. *Concurr. Comput. Pract. Exp.*, 23(2):158–171, 2011.

[JF20] Jan-Keno Janssen and Martin Fischer. Schön schweben. *c't*, 19/2020:52–53, Aug 2020.

[JWP05] Stefan Jeschke, Michael Wimmer, and Werner Purgathofer. Image-based representations for accelerated rendering of complex scenes. In Yiorgos

Chrysanthou and Marcus A. Magnor, editors, *26th Annual Conference of the European Association for Computer Graphics, Eurographics 2005 - State of the Art Reports, Dublin, Ireland, August 29 - September 2, 2005*, pages 1–20. Eurographics Association, 2005.

[MBG+12] Jean-Eudes Marvie, Cyprien Buron, Pascal Gautron, Patrice Hirtzlin, and Gaël Sourimant. GPU shape grammars. *Comput. Graph. Forum*, 31(7-1):2087–2095, 2012.

[MES14] Pedram Mohammadi, Abbas Ebrahimi-Moghadam, and Shahram Shirani. Subjective and objective quality assessment of image: A survey. *CoRR*, abs/1406.7799, 2014.

[MWH+06] Pascal Müller, Peter Wonka, Simon Haegler, Andreas Ulmer, and Luc Van Gool. Procedural modeling of buildings. *ACM Trans. Graph.*, 25(3):614–623, 2006.

[Sch97] Gernot Schaufler. Nailboards: A rendering primitive for image caching in dynamic scenes. In Julie Dorsey and Philipp Slusallek, editors, *Rendering Techniques '97, Proceedings of the Eurographics Workshop in St. Etienne, France, June 16-18, 1997*, Eurographics, pages 151–162. Springer, 1997.

[Sch98] Gernot Schaufler. Image-based object representation by layered impostors. In J. M. Shieh and Shi-Nine Yang, editors, *Proceedings of the ACM Symposium on Virtual Reality Software and Technology, VRST 1998, Taipei, Taiwan, November 2-5, 1998*, pages 99–104. ACM, 1998.

[SG71] George Stiny and James Gips. Shape grammars and the generative specification of painting and sculpture. In Charles V. Freiman, John E. Griffith, and Jack L. Rosenfeld, editors, *Information Processing, Proceedings of IFIP Congress 1971, Volume 2 - Applications, Ljubljana, Yugoslavia, August 23-28, 1971*, pages 1460–1465. North-Holland, 1971.

[SKK+14a] Markus Steinberger, Michael Kenzel, Bernhard Kainz, Jörg Müller, Peter Wonka, and Dieter Schmalstieg. Parallel generation of architecture on the GPU. *Comput. Graph. Forum*, 33(2):73–82, 2014.

[SKK+14b] Markus Steinberger, Michael Kenzel, Bernhard Kainz, Peter Wonka, and Dieter Schmalstieg. On-the-fly generation and rendering of infinite cities on the GPU. *Comput. Graph. Forum*, 33(2):105–114, 2014.

[SLG19] Thomas Schauppenlehner, Konstantin Lux, and Christoph Graf. Effiziente großflächige interaktive landschaftsvisualisierungen im kontext des ausbaus erneuerbarer energie–das potenzial freier geodaten für die entwicklung interaktiver 3d-visualisierungen. *AGIT Journal für Angewandte Geoinformatik*, 5:172–182, 2019.

[Str15] Allen Stroud. Developing elite dangerous. *Foundation*, (120):78–88, 2015.

[TN21] Emma R Tait and Ingrid L Nelson. Nonscalability and generating digital outer space natures in no man's sky. *Environment and Planning E: Nature and Space*, page 251484862110007, mar 2021.

[VWB+21] Abhinau K. Venkataramanan, Chengyang Wu, Alan C. Bovik, Ioannis Katsavounidis, and Zafar Shahid. A hitchhiker's guide to structural similarity. *IEEE Access*, 9:28872–28896, 2021.

[WBSS04] Zhou Wang, Alan C. Bovik, Hamid R. Sheikh, and Eero P. Simoncelli. Image quality assessment: from error visibility to structural similarity. *IEEE Trans. Image Process.*, 13(4):600–612, 2004.

[WSB03] Z. Wang, E.P. Simoncelli, and A.C. Bovik. Multiscale structural similarity for image quality assessment. In *The Thrity-Seventh Asilomar Conference on Signals, Systems Computers, 2003*, volume 2, pages 1398–1402 Vol.2, 2003.

[WWSR03] Peter Wonka, Michael Wimmer, François Sillion, and William Ribarsky. Instant architecture. *ACM Trans. Graph.*, 22(3):669–677, July 2003.