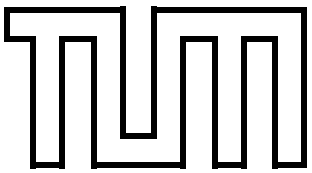


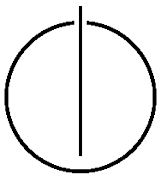
FAKULTÄT FÜR INFORMATIK

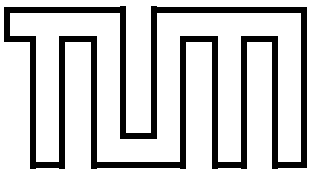
TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Computer Science

Assisted Object Placement

Andreas Kirsch





FAKULTÄT FÜR INFORMATIK

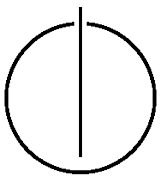
TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Computer Science

Assisted Object Placement

Unterstützte Platzierung von Objekten

Author:	Andreas Kirsch
Supervisor:	Prof. Dr. Westermann
Advisor:	Matthäus G. Chajdas
Date:	November 15, 2012



I assure the single handed composition of this master's thesis only supported by declared resources.

Munich, November 19, 2012 Andreas Kirsch

Acknowledgments

This master's thesis would not have been possible without my advisor Matthäus G. Chajdas, who found time to answer my questions at the most unusual times of day and was always a helpful force.

I would also like to thank my friends, in particular—but no particular order—An, Andreas, Matthias, Jan, Julia, Xin, Ben, Stefan, Alex and Felix, for listening and discussing various problems with me, giving me advice and being fun to spend time with; and my other friends and family for having patience: they have not seen me much lately.

In addition, I am also thankful for helpful comments from Sylvain Lefebvre, Prof Hoffman and Prof Cremers.

Abstract

Populating large scenes with objects is a time-consuming task. In many cases, the objects fulfill a purely decorative purpose, so the actual placement is not critical. We present an assistant system for level designers which helps them place such objects in a scene. The core idea is to sample a scene using many small probes that each store a bit of information about the local environment and match these with a dynamically generated database to suggest suitable candidates for placement. Additionally, we use the spatial relationships between objects to learn local layouts of the scene and refine the suggestions.

Our placement algorithms are fast enough to run in real-time on complete game levels from a shipped game. To demonstrate this, we develop a small level editor that shows the abilities of our algorithms.

Zusammenfassung

Das Platzieren von Objekten in großen Szenen ist ein zeitaufwändiger Prozess. In vielen Fällen haben die Objekte nur eine dekorative Funktion; somit kommt es nicht auf ihre genaue Position an. Wir stellen ein Assistenz-System vor, dass Level Designern dabei hilft, solche Objekte in einer Szene zu platzieren. Unser Ansatz besteht darin, eine Szene mit Hilfe vieler, kleiner Proben zu sampeln, wobei jede Probe ein kleinen Teil der lokalen Umgebung speichert. Wir vergleichen diese mit dem Inhalt einer dynamisch generierten Datenbank, um die am besten passenden Kandidaten für eine Platzierung zu bestimmen. Des Weiteren nutzen wir die räumliche Beziehung der Objekte aus, um ihre lokale Anordnung zu erlernen und unsere Vorschläge zu verfeinern.

Unser Algorithmus ist schnell genug, um in Echtzeit mit den vollständigen Levels eines veröffentlichten Spieles zu arbeiten. Um dies zu demonstrieren, entwickeln wir einen kleinen Level-Editor, der die Möglichkeiten unserer Algorithmen beweist.

Contents

Acknowledgements vii

Abstract ix

Zusammenfassung xi

Contents xiii

1 Introduction 1
- 1.1 Motivation . . . 1
- 1.2 Background . . . 2
- 1.3 Goals . . . 2
- 1.4 A real-world test case: Shadowgrounds Survivor . . . 2
- 1.5 Probabilistic model . . . 4
  - 1.5.1 Formulation of the problem . . . 5
  - 1.5.2 Context maps and importance weights . . . 8
  - 1.5.3 Importance weights from information theory . . . 11

2 Algorithms 15
- 2.1 Overview . . . 15
- 2.2 Matching using probes . . . 17
  - 2.2.1 Motivation . . . 17
  - 2.2.2 Probe generation and probe samples . . . 18
  - 2.2.3 Finding candidates . . . 22
- 2.3 Matching using neighborhood distances . . . 26
  - 2.3.1 Motivation . . . 26
  - 2.3.2 Distance sets . . . 27
  - 2.3.3 Implementation . . . 31
- 2.4 Combined matching . . . 36

3 Implementation **37**
3.1 Demonstration application . . . 37
3.2 Interaction with Shadowgrounds Survivor . . . 40
3.2.1 Asset pipeline . . . 40
3.2.2 Model and level data . . . 41
3.2.3 Exporters . . . 42
3.2.4 Rendering . . . 45
3.3 Voxelizer . . . 45
3.3.1 Basic implementation . . . 45
3.3.2 Conservative rasterization . . . 47
3.4 Probe directions . . . 51
3.4.1 Possible direction sets . . . 51
3.4.2 Determining probe directions . . . 52
3.5 Probe matching . . . 53

4 Results **59**
4.1 Validation using artificial scenes . . . 59
4.1.1 Neighborhood context . . . 59
4.1.2 Probe context . . . 63
4.2 Validation using Shadowgrounds Survivor . . . 66
4.2.1 Neighborhood context . . . 69
4.2.2 Probe context . . . 71
4.2.3 Combined matching . . . 74
4.3 Performance . . . 75

5 Conclusion **77**

Bibliography **79**

Chapter 1

Introduction

1.1 Motivation

Research work in computer graphics is applied in many areas. One of the most prominent is the entertainment industry: computer games are an important beneficiary of new developments in computer graphics.

The most expensive part of any game is content production. Hundreds of man years are required to create a modern AAA computer game. This also shows in the production costs for current titles.

However, even though this is a major cost factor, only few parts of content production are automated at the moment. The reason for this is that all final decisions have to be made by the artists, and most research only looks into closed solutions for special problems of content generation. Most of them cannot be incorporated into an artist's workflow easily or are difficult to adjust.

Current games use lots of assets in complex levels. Many level designers work on them in turns without perfectly knowing the level or all used assets. Especially small decorative objects, commonly called props, are a burden for level designers because many of them have to be placed throughout the scene. It is difficult to keep track of them as their exact position is not important, and while it seems that this task should be easy to automate, it still involves a lot of manual labor. We believe that a lot of time can be saved by developing a new algorithm to help designers place props in levels.

Assisted Object Placement can drastically reduce the time required to populate a game level. Instead of having to work with potentially hundreds of assets that could be present in a level, our algorithm reduces the candidate set dramatically and provides the designer a list of candidate objects. This is particularly important as new level designers immediately get a hint of how objects have already been placed in the level, and it reduces the time required to search for matching assets.

We analyze a level to learn the surroundings where a particular object type is placed as well as the spatial relationship between objects. No additional input besides the objects themselves and their geometry is necessary for our algorithm.

1.2 Background

Historically, L-systems and shape grammars have been used to generate models and levels procedurally. This is popular for generating city layouts [PM01, VKW+12]. At the same time, specifying the rules for them is not easy, and it is even more difficult to create rules that target a certain outcome. [TLL+11] shows a way to guide such systems to fulfill certain constraints using optimization techniques on the space of possible outputs of the algorithm. However, level designers do not want to interrupt their work and cannot be expected to develop rules for content generation.

Recent research looks at example-based generation of scenes [FRS+12]. It requires the creation of examples by hand however. [YYW+12] uses random Markov processes to create open-world layouts, but also expects constraints to be specified beforehand.

None of these approaches attempts to add to existing scenes. Rather, all of them generate totally new content, and they consequently cannot be used to aid artists with populating levels. Furthermore, they cannot adapt quickly to changing scenes and are not designed to learn unsupervised.

[CLS10] uses geometric properties of surfaces in a scene and a clever matching algorithm to learn from existing content and to automatically suggest textures for new geometry. The users only have to specify few parameters to adapt the algorithm to their needs, and it works well in different scenarios. The algorithm creates an easy-to-navigate list with suggestions that the user can select from.

We use this work as basis for Assisted Object Placement to create suggestions for object placement without any need for additional semantic information, manually created example data, or pre-specified rules and constraints.

1.3 Goals

The goal of this thesis is to develop an algorithm for automatic object placement that can learn how to place objects from an existing scene and that requires no additional information about the objects or the scene. It should only use the geometry of objects in the scenes and their spatial relationships. In addition, it should:

- require little user interaction and offer a simple interface;
- work with the limited data provided by a single scene;
- learn from new objects as they are added or removed; and
- be robust and work in a real-world test case.

The last point is especially important. Artificial test cases are necessary to verify scientific hypotheses and understand algorithms, but an aid like Assisted Object Placement is only useful if it works in a real, practical environment.

1.4 A real-world test case: Shadowgrounds Survivor

We have decided to use the game Shadowgrounds Survivor from *Frozenbyte* as a real-world example. The game has been released in 2007 and was open sourced in 2011. In the following section we are going to give some details about the game and describe the reasons why we chose it.

Factors for choosing the game

When we started thinking about which game we could use, some constraints developed naturally. Namely the game should:

(a) not require any license fees,
(b) be open source (both game and tools),
(c) allow a fast and simple export of level and model data, and
(d) be old enough that unoptimized prototype code will be able to render and process game data sufficiently fast on a modern workstation.

These considerations are obviously interlocked: for example, (d) facilitates both (c) and (b). The game should be open source because we do not want to reverse engineer any formats. In addition, open-source tools hopefully allow us to build on an already existing codebase. The age of the game is an important and often overlooked factor: with a recently published, high-end game you cannot run experiments as fast as with a game that has targeted older hardware. Moreover, we have found that its levels should:

- use a high number of distinguishable props,
- be set in different environments,
- be diverse internally, and
- be easy to navigate.

The level constraints are derived from the aims of Assisted Object Placement: we can only expect meaningful validation results if the levels are diverse and we have lots of props that are used in different places and contexts. It helps the quality of our validation if we have levels in varying environments since then we can be sure that our approach works in more cases. Last, if the levels were not easy to navigate, it would become unnecessarily time-consuming to verify the results.

Shadowgrounds Survivor

Shadowgrounds Survivor is a third-person action shooter that lets a player fight for survival against alien invaders. The players control their characters from an overhead view and navigate through open environments. Its levels use many props and are set in industrial environments, snowy landscapes and lave-filled caves. See figure 1.1 on the following page.

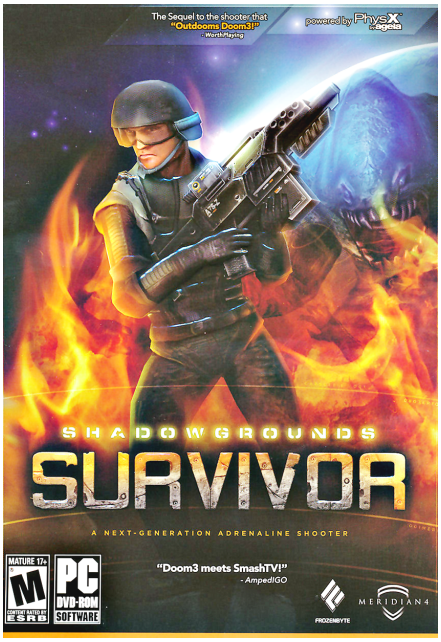
Comparison to other open-sourced games

We have looked at other games to evaluate if they are usable for validation, but we have decided against them:

Doom 3 has been open sourced in 2011 as well, but its levels are difficult to navigate in comparison, and it does not use as many environments.

Nexuiz, **Tremulous**, **and the Quake series games** have been open sourced, but are older and do not yet use as many props as the newer Shadowgrounds Survivor.

UFO: **Alien Invasion** is comparable to Shadowgrounds Survivor and has many prop models as well, but it lacks the polish Shadowgrounds Survivor has.



(a) Shadowgrounds Survivor's packaging

(b) A screenshot showing various props in the level `marine01_wakeup`

Figure 1.1: Shadowgrounds Survivor

1.5 Probabilistic model

An underlying probabilistic model can make algorithms more robust. Moreover, it can give sound answers when design questions arise. In the following, we are going to look at a small probabilistic framework that we are going to motivate using basic theorems of probability theory.

Sources. The notation and the reasoning are heavily inspired by [Bis06]. The parts of probability theory we refer to are part of any introduction to probability theory, see for example [SS] or [Ros07].

Basic definitions

Instead of using the ambiguous term "object", we are going to use the term **instance** for an instance of a certain **model** in the scene. Whenever we talk of a model in this section, we identify it with the set of all its instances. Symbolically if we have a model $\mathcal{M}$ with instances $\mathbf{I_1}$, $\mathbf{I_2}$, …, $\mathbf{I_m}$ we identify

$$\mathcal{M} \equiv \{\mathbf{I_1}, \mathbf{I_2}, \ldots, \mathbf{I_m}\}. \tag{1.2}$$

For the family of all models, we are going to use the symbol $\mathbb{M}$, and, likewise, the symbol $\mathbb{I}$ for the family of all instances. By definition, $\mathbb{M}$ is a partition of $\mathbb{I}$:

$$\mathbb{I} = \biguplus_{\mathcal{M}\in\mathbb{M}} \mathcal{M} \tag{1.3}$$

Scene and contexts

We need to define how instances relate to a scene. For a given **context space** $\mathfrak{C}$, we will specify a **context map** C that maps an instance to its context:

$$\mathsf{C}\colon \mathbb{I} \longrightarrow \mathfrak{C}. \tag{1.4}$$

A **context** describes the scene that surrounds an instance. For our probabilistic model, it is the "window" into the scene. It defines how we experience the relationship between instances and the scene, so choosing the right context is important.

To be able to determine whether a model is a good candidate at a certain position in the scene, we must be able to map positions to a context. We use the symbol $\mathbb{U}$ for the scene coordinate space. Usually we have $\mathbb{U} = \mathbb{R}^3$. We "overload" the context map to provide contexts for scene coordinates as well:

$$\mathsf{C}\colon \mathbb{U} \longrightarrow \mathfrak{C}. \tag{1.5}$$

1.5.1 Formulation of the problem

Using probability theory, we can express the problem of determining the model $\mathcal{M}^\star$ that fits best to a certain position $\mathbf{p}$ in the scene as:

$$\mathcal{M}^\star = \arg\max_{\mathcal{M}\in\mathbb{M}} P\big(\mathcal{M} \,\big|\, C = \mathsf{C}(\mathbf{p})\big), \tag{1.6}$$

where C is the random variable denoting the context. When there is no ambiguity about the random variable, we write:

$$\mathcal{M}^\star = \arg\max_{\mathcal{M}\in\mathbb{M}} P\big(\mathcal{M} \,\big|\, \mathsf{C}(\mathbf{p})\big). \tag{1.7}$$

The equations above just express that we want to find the model with the highest probability for a certain position in the scene. This might seem strange at first, but makes sense when we consider it from a Bayesian point of view. For example, if our algorithm is unsure which model to suggest from a set of possible candidates, a probability value is the best way to express this uncertainty.

We use a context map because a position itself does not have any meaning. Actually, it is an arbitrary value since different coordinate spaces could be chosen. As a result, we map a position to a context that describes what we "see" at that position, which we can then relate to prior knowledge about models and the contexts in which their instances have been placed before.

Actually, we do not just want to find the best model for a certain position: we want to find a candidate ranking of all models, which we can present to the level designers to choose the one they prefer. So, put differently, we want to find a non-strict weak ordering $\mathcal{M}^{\star n} \le \cdots \le \mathcal{M}^{\star 1}$ of all models, so that

$$P\big(\mathcal{M}^{\star n} \,\big|\, \mathsf{C}(\mathbf{p})\big) \le \cdots \le P\big(\mathcal{M}^{\star 1} \,\big|\, \mathsf{C}(\mathbf{p})\big). \tag{1.8}$$

We do not know yet how to extend the context map C to models, so the question is: how can we define C for a model $\mathcal{M}$ based on its instances? First, we can use the definition of conditional

probability[1] to obtain

$$P\big(\mathcal{M} \mid \mathsf{C}(\mathbf{p})\big) = \frac{P(\mathsf{C}(\mathbf{p}), \mathcal{M})}{P(\mathsf{C}(\mathbf{p}))}. \tag{1.9}$$

We can expand the numerator further using the sum rule to

$$P(\mathsf{C}(\mathbf{p}), \mathcal{M}) = \sum_{\mathbf{I} \in \mathcal{M}} P(\mathsf{C}(\mathbf{p}), \mathbf{I}, \mathcal{M}), \tag{1.10}$$

and then using the product rule to

$$= \sum_{\mathbf{I} \in \mathcal{M}} P\big(\mathsf{C}(\mathbf{p}) \mid \mathbf{I}, \mathcal{M}\big) P\big(\mathbf{I} \mid \mathcal{M}\big) P(\mathcal{M}). \tag{1.11}$$

To simplify $P\big(\mathsf{C}(\mathbf{p}) \mid \mathbf{I}, \mathcal{M}\big)$, we need to investigate a property of random variables and events.

Total dependence

Definition 1.2. A random variable X is **totally dependent** on a random variable Y, if

$$\forall x \in \operatorname{dom} X, y \in \operatorname{dom} Y : P\big(X = x \mid Y = y\big) \in \{0, 1\}. \tag{1.12}$$

Lemma 1.3. *As $\sum_X P\big(X \mid Y\big) = 1$, we see that, for every y, there exists exactly one x_y with $P\big(X = x_y \mid Y = y\big) = 1$ and $P\big(X = x \mid Y = y\big) = 0$ for all $x \neq x_y$; that is*

$$P\big(X = x \mid Y = y\big) = \mathbb{1}_{\{x = x_y\}}. \tag{1.13}$$

From this, we directly obtain

$$P(Y = y) = \sum_{x \in \operatorname{dom} X} P(X = x, Y = y) = P(X = x_y, Y = y). \tag{1.14}$$

Proposition 1.4. *If X is totally dependent on Y and A is a random variable, the following identity holds:*

$$\forall y \in \operatorname{dom} Y : P(A, Y = y) = P(A, X = x_y, Y = y). \tag{1.15}$$

Proof. We begin with the sum rule:

$$P(A, Y) = \sum_X P(A, X, Y). \tag{1.16}$$

Again, with the product rule, we obtain

$$\begin{aligned} &= \sum_X P\big(A \mid X, Y\big) P\big(X \mid Y\big) P(Y) \\ &= \sum_{x \in \operatorname{dom} X} P\big(A \mid X = x, Y = y\big) P\big(X = x \mid Y = y\big) P(Y = y) \\ &= \sum_{x \in \operatorname{dom} X} P\big(A \mid X = x, Y = y\big) \mathbb{1}_{\{x = x_y\}} P(Y = y) \\ &= P\big(A \mid X = x_y, Y = y\big) P(Y = y). \end{aligned} \tag{1.17}$$

[1] we could use Bayes' theorem here as well

Finally, we apply Bayes' theorem and equation (1.14) to get

$$\begin{aligned} &= \frac{P(A, X = x_y, Y = y)}{P(X = x_y, Y = y)} P(Y = y) \\ &= \frac{P(A, X = x_y, Y = y)}{P(X = x_y, Y = y)} P(X = x_y, Y = y) \\ &= P(A, X = x_y, Y = y), \end{aligned} \tag{1.18}$$

and are done. □

This works for events, ie sets, too, when we work with its characteristic function[2].

Definition 1.5. An event S is called totally dependent on an event T, if

$$P\big(S \,\big|\, T\big) = 1. \tag{1.19}$$

Remark. We could define this similar to definition 1.2 and use $\bullet \in \{0, 1\}$, but this just adds clutter similar to lemma 1.3: either $P\big(S \,\big|\, T\big) = 1$ or $P\big(\overline{S} \,\big|\, T\big) = 1$. It is cleaner to just require $P\big(S \,\big|\, T\big) = 1$.

Corollary 1.7.

1. *If an event S is totally dependent on an event T, then*

$$P(T) = P(S, T). \tag{1.20}$$

2. *If an event S is totally dependent on an event T and A is a random variable, then*

$$P(A, T) = P(A, S, T). \tag{1.21}$$

3. *If an event S is totally dependent on an event T and A is a random variable, then*

$$P\big(A \,\big|\, X, Y\big) = P\big(A \,\big|\, Y\big) \tag{1.22}$$

Proof. Equation (1.20) and equation (1.21) follow directly:

$$P(T) = P(\lambda_T = 1) = P(\lambda_S = 1, \lambda_T = 1) = P(S, T) \tag{1.23}$$

$$P(A, T) = P(A, \lambda_T = 1) = P(A, \lambda_S = 1, \lambda_T = 1) = P(A, S, T) \tag{1.24}$$

For equation (1.22), we simply apply the definition of conditional probability, and equations (1.20) and (1.21):

$$\begin{aligned} P\big(A \,\big|\, X, Y\big) &= \frac{P(A, X, Y)}{P(X, Y)} \\ &= \frac{P(A, Y)}{P(Y)} \\ &= P\big(A \,\big|\, Y\big). \end{aligned} \tag{1.25}$$

□

[2] The characteristic function λ_S of a set S is defined as $\lambda_S(s) = \mathbb{1}_{\{s \in S\}}$.

Simplification: continued

From equation (1.3), we see that $\mathcal{M}$ is totally dependent on all its instances $\mathbf{I} \in \mathcal{M}$ (and $\overline{\mathcal{M}}$ on all $\mathbf{I} \in \overline{\mathcal{M}}$). We can finally simplify

$$P(\mathsf{C}(\mathbf{p}), \mathcal{M}) = \sum_{\mathbf{I} \in \mathcal{M}} P\big(\mathsf{C}(\mathbf{p}) \,\big|\, \mathbf{I}, \mathcal{M}\big) P\big(\mathbf{I} \,\big|\, \mathcal{M}\big) P(\mathcal{M}) \tag{1.26}$$

$$= \sum_{\mathbf{I} \in \mathcal{M}} P\big(\mathsf{C}(\mathbf{p}) \,\big|\, \mathbf{I}\big) P\big(\mathbf{I} \,\big|\, \mathcal{M}\big) P(\mathcal{M}). \tag{1.27}$$

All in all, we obtain

$$P\big(\mathcal{M} \,\big|\, \mathsf{C}(\mathbf{p})\big) = \sum_{\mathbf{I} \in \mathcal{M}} \frac{P\big(\mathsf{C}(\mathbf{p}) \,\big|\, \mathbf{I}\big) P\big(\mathbf{I} \,\big|\, \mathcal{M}\big) P(\mathcal{M})}{P(\mathsf{C}(\mathbf{p}))}. \tag{1.28}$$

We can write

$$P\big(\mathcal{M} \,\big|\, \mathsf{C}(\mathbf{p})\big) \propto \sum_{\mathbf{I} \in \mathcal{M}} P\big(\mathsf{C}(\mathbf{p}) \,\big|\, \mathbf{I}\big) P\big(\mathbf{I} \,\big|\, \mathcal{M}\big) P(\mathcal{M}), \tag{1.29}$$

to express that the left term is proportional to the right term. We use this often because the denominator can be found using the sum rule and is thus implicitly defined by the numerator.

If we use that $P\big(\mathbf{I} \,\big|\, \mathcal{M}\big) P(\mathcal{M}) = P(\mathbf{I}, \mathcal{M}) = P(\mathbf{I})$ for $\mathbf{I} \in \mathcal{M}$, we can write

$$P\big(\mathcal{M} \,\big|\, \mathsf{C}(\mathbf{p})\big) = \sum_{\mathbf{I} \in \mathcal{M}} \frac{P\big(\mathsf{C}(\mathbf{p}) \,\big|\, \mathbf{I}\big) P(\mathbf{I})}{P(\mathsf{C}(\mathbf{p}))}$$

$$= \sum_{\mathbf{I} \in \mathcal{M}} P\big(\mathbf{I} \,\big|\, \mathsf{C}(\mathbf{p})\big). \tag{1.30}$$

Interpretation

With equation (1.29), we can interpret $P(\mathcal{M})$ as prior knowledge and $P\big(\mathbf{I} \,\big|\, \mathcal{M}\big)$ as importance of a specific instance of a model. Equation (1.30) shows the linear dependence of the probability of a model on its instances.

1.5.2 Context maps and importance weights

The context maps are never going to be simple values, so the question arises how to compute $P\big(\mathsf{C}(\mathbf{x}) \,\big|\, \mathcal{M}\big)$ when $\mathsf{C}(\mathbf{x})$ itself is a function of some parameters. For example, if we were to sample the distance to surrounding geometry (again in $\mathbb{U} = \mathbb{R}^2$), the context space would consist of functions that map from the unit circle to a distance:

$$\mathfrak{C} = \left\{c \colon S^1 \longrightarrow \mathbb{R}_0^+\right\}. \tag{1.31}$$

The points on the unit circle represent the different directions we can look into at a certain point. The corresponding context map maps points onto these functions:

$$\mathsf{C} \colon \mathbb{R}^2 \longrightarrow S^1 \longrightarrow \mathbb{R}_0^+. \tag{1.32}$$

We can also interpret the context map as function that maps a point and a direction to a distance:

$$\mathsf{C}\colon \mathbb{R}^2 \times S^1 \longrightarrow \mathbb{R}_0^+. \tag{1.33}$$

Both notations are equivalent.

Either way, we have to create a distribution for each context using our sample data. We have two options:

1. we can treat each context atomically; or
2. we can look at the values of a context over its parameter space and deal with these values independently.

For instance, if we have samples $c_1, \ldots, c_m$ for a model $\mathcal{M}$, then (1) could mean choosing a distance metric d in the context space and a weight function ω so that we can set

$$P\big(C = c \,\big|\, \mathcal{M}\big) \propto \sum_i \omega(d(c, c_i)), \tag{1.34}$$

with $\propto$ instead of $=$ because the right term is not normalized.

One aim of Assisted Object Placement is to allow for cases where contexts are matched partially by different instances: if there is a blue wall to the left of a model for half of its instances and a red wall to the right for the other half, then the model should be suggested for a context with both a blue and a red wall with a certainty as if there had been instances with a blue wall to the left of them and a red wall to the right of them. Using metrics and weight functions from the start makes it difficult to check whether we fulfill this goal, so let us examine (2).

(2) means that we split the context into **sub-contexts** and treat them independently. For example, if we use a simplified version of the context space above with only finitely many elements:

$$\mathfrak{C} := \{c\colon N \longrightarrow \mathbb{R}_0^+\} \text{ with } N := \{1, \ldots, n\}, \tag{1.35}$$

we would separately look at each probability $P\big(C(i) \,\big|\, \mathcal{M}\big)$, for $i \in N$, where we use $\mathsf{C}_i(\mathbf{x})$ as shorthand for $\mathsf{C}(\mathbf{x})(i)$, and similarly C_i for $C(i)$.

But now we are left with individual probabilities—how can we combine them? There is no straight-forward answer. Since we are in a probabilistic setting and are observing independent variables, we could try to use the following ansatz:

$$P\big(C \,\big|\, \mathcal{M}\big) = \prod_i P\big(C(i) \,\big|\, \mathcal{M}\big). \tag{1.36}$$

However, this gives too much importance to sub-contexts that do not match. If one does not match the sample data at all, or just "a bit", the probability of the whole context is going to be very small, even if all other sub-contexts match the sample data very well.

A better approach is to view the sub-contexts as each telling its "opinion" about the context. Put differently, the probability $P\big(C(i) \,\big|\, \mathcal{M}\big)$ is a guess of the i-th sub-context regarding the probability $P\big(C \,\big|\, \mathcal{M}\big)$. We can assign to each sub-context a probability for how important a sub-context's opinion is for the context[3]. Then, the first moment, the expectation, over the sub-contexts' probabilities will give the probability of the context.

[3]Picture this probability as being obtained from the following thought experiment: we know $P\big(C \,\big|\, \mathcal{M}\big)$ and we look at how often $P\big(C(i) \,\big|\, \mathcal{M}\big)$ matches it.

Put differently, we can assign each $P\big(C(i) \,\big|\, \mathcal{M}\big)$ an **importance weight** $\Phi\big(i \,\big|\, C, \mathcal{M}\big)$ and then use

$$P\big(C \,\big|\, \mathcal{M}\big) = \frac{\sum_i \Phi\big(i \,\big|\, C, \mathcal{M}\big) P\big(C(i) \,\big|\, \mathcal{M}\big)}{\sum_i \Phi\big(i \,\big|\, C, \mathcal{M}\big)} \tag{1.37}$$

$$\propto \sum_i \Phi\big(i \,\big|\, C, \mathcal{M}\big) P\big(C(i) \,\big|\, \mathcal{M}\big). \tag{1.38}$$

$\Phi\big(i \,\big|\, C, \mathcal{M}\big)$ cannot be used like a probability measure because it is not normalized. However, we can normalize it to obtain the **normalized importance weight** $\hat{\Phi}$:

$$\hat{\Phi}\big(i \,\big|\, C, \mathcal{M}\big) = \frac{\Phi\big(i \,\big|\, C, \mathcal{M}\big)}{\sum_i \Phi\big(i \,\big|\, C, \mathcal{M}\big)}. \tag{1.39}$$

$\hat{\Phi}\big(\bullet \,\big|\, C, \mathcal{M}\big)$ has the benefit that we can use all the laws of probability theory to build more complex importance weights. We can continue to use a probabilistic approach for constructing them as above, or we can use heuristics.

General notation

Given a context space $\mathfrak{C}$with

$$\mathfrak{C} = \{c \colon V_1 \times V_2 \ldots \longrightarrow \mathfrak{X},\ (v_1, v_2, \ldots) \mapsto c(v_1, v_2, \ldots)\}\,, \tag{1.40}$$

with a corresponding context map C, we write $\Phi\big(v_1, v_2, \ldots \,\big|\, C, \mathcal{M}\big)$ for the importance weight of these parameters and $P\big(C(v_1, v_2, \ldots) \,\big|\, \mathcal{M}\big)$ for the probability distribution of the respective sub-context.

We again define

$$\hat{\Phi}\big(v_1, v_2, \ldots \,\big|\, C, \mathcal{M}\big) = \frac{\Phi\big(v_1, v_2, \ldots \,\big|\, C, \mathcal{M}\big)}{\sum_{v_1, v_2, \ldots} \Phi\big(v_1, v_2, \ldots \,\big|\, C, \mathcal{M}\big)} \tag{1.41}$$

as normalized importance weight. The probability $P\big(C \,\big|\, \mathcal{M}\big)$ can then be calculated as expectation over $\hat{\Phi}\big(\bullet \,\big|\, C, \mathcal{M}\big)$:

$$\begin{aligned} P\big(C \,\big|\, \mathcal{M}\big) &= E_{\hat{\Phi}}\left[P\big(C(v_1, v_2, \ldots) \,\big|\, \mathcal{M}\big)\right] \\ &= \sum_{v_1, v_2, \ldots} \hat{\Phi}\big(v_1, v_2, \ldots \,\big|\, C, \mathcal{M}\big)\, P\big(C(v_1, v_2, \ldots) \,\big|\, \mathcal{M}\big). \end{aligned} \tag{1.42}$$

This is kind of argument is related to importance sampling, which is covered in detail by [PH04, chapters 14 and 15].

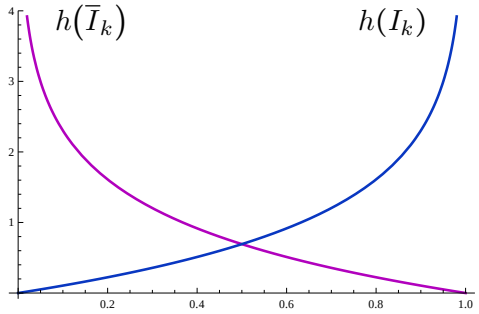


Figure 1.44: Message lengths for different values of $P(I_k)$. $\overline{I}_k$ is the complementary event of I_k.

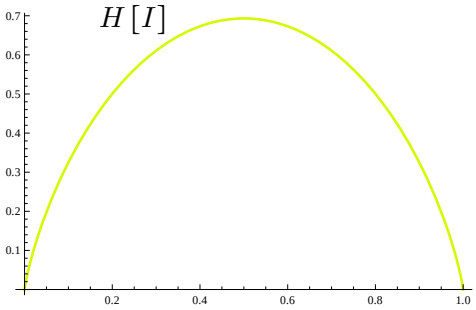


Figure 1.45: Entropy of a Bernoulli distribution plotted for different values of its parameter p, which is the probability of a positive outcome.

1.5.3 Importance weights from information theory

Without prior knowledge the best choice for an importance weight is a uniform distribution over the parameter space. However, if we have prior knowledge, a measure that quantifies the "surprise" with regard to the value of a certain sub-context could constitute a good importance weight.

To see this, we can use the following thought experiment: If we try to determine which objects fit best in a universe in which everything has the color blue, the fact that an object has the color blue adds no information at all as this is true for all objects. However, if an object were of a different color, this would be a most astounding observation and be a very important piece of information as it quite probably should disqualify the object from existing in that universe at all. Likewise, if a certain color almost never appears in a scene, the fact that it is missing is not of much importance, but, when it occurs, it is all the more interesting.

To sum it up, we notice that the more unexpected an outcome is, the more information content it conveys. We can quantify this by looking at information theory.

Information theory

Each piece of information, or short **message**, I_k has a probability $P(I_k)$. An unexpected message thus has a low probability. Each possible message is assigned a **message length** h:

$$h(I_k) := -\ln P(I_k). \tag{1.43}$$

The message length specifies the optimal length in **nats** (natural bits) of a message[4]. See figure 1.44 for a plot of $h(I_k)$ for different values of $P(I_k)$. The message length is closely related to the (information) entropy H:

$$H[I] := \sum_k P(I_k)\, h(I_k), \tag{1.46}$$

[4]To measure the length in bits, you would have to use $\log_2$.

which can be also used to prove that this definition of the message length is the optimal one. Figure 1.45 on the preceding page shows a plot of the entropy function for a Bernoulli distribution[5].

See [Bis06, chapter 1] for a more detailed introduction.

Approximation of the message length using relative frequencies

To calculate the message length, we need to know the probability of a message. We do not know the actual probability distribution, but we can approximate it with prior knowledge by using its relative frequency. Naturally, such an approximation causes issues, and in this case there are two:

1. if a message occurs, that has never occurred before, its relative frequency is 0, and its length will tend towards infinity; and
2. if a message always occurs, then its relative frequency is 1, and its length is 0.

Generally, (1) can be avoided by using a correction value α that is used as probability for yet-unknown messages. The other relative frequencies have to be adjusted accordingly to keep the sum of all frequencies normalized. α should not be greater than the smallest computed frequency. Otherwise, results could be distorted too much.

If we have n example messages as prior knowledge and a test message whose length we want to determine, and none of the example messages matches the test message, we can simply calculate the total frequency of the example set including the test message. Then the test message has a frequency of $\frac{1}{n+1}$, and all other frequencies automatically become smaller.

(2) would not be an issue if we used the actual probability distribution: it makes sense to discard such messages because messages that always occur offer no information. However, since we only know the relative frequency of a small dataset, it is a bad idea to totally discard the information of any message.

Instead, we can once more use a correction value β. We note that all $n+1$ observed messages—n example messages and the test message—had the same outcome and set $\beta := \frac{n+1}{n+2}$.

Implementation for a Bernoulli sub-context

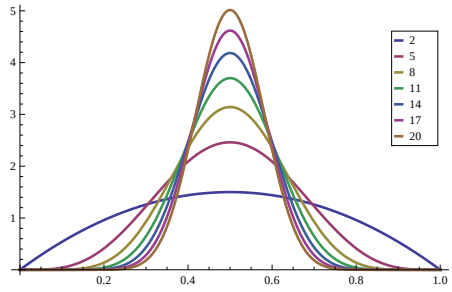
In a Bernoulli sub-context, that is a sub-context which has a Bernoulli distribution, we have only two possible messages. Our correction values can be interpreted as starting with prior knowledge that already contains both possible outcomes once. The relative frequency given n samples and l positive outcomes is then $\frac{l+1}{n+2}$.

This is similar to using a Beta distribution as prior distribution for the sub-context with $a = 1$ and $b = 1$ initially, and using its mean as probability of the Bernoulli sub-context. See [Bis06, chapter 2] for more details. Figure 1.48a on the next page shows Beta distributions $\mathrm{Beta}\big(\bullet \mid a = n, b = n\big)$ with values for n ranging between 2 and 20. With increasing sample count $a + b$, we see that the graphs converge towards the Dirac delta function $\delta_{0.5}$, which has its peak at 0.5.

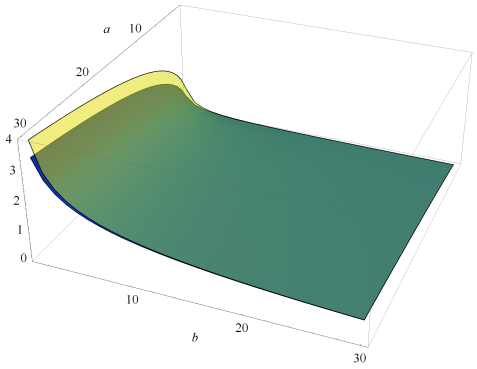
This interpretation allows us to view the message length as a random variable over the prior distribution and estimate it using its expectation value:

$$h(X) = E_{\mathrm{Beta}(\bullet|a,b)}\left[h\big(X \mid \mathrm{Beta}(\bullet|a,b)\big)\right]. \tag{1.47}$$

[5] A Bernoulli distribution is a "binary" distribution: it describes a random variable with exactly two outcomes. The probability of a positive outcome is denoted using the parameter p.



(a) Plots of $\mathrm{Beta}(\bullet \mid a = n, b = n)$ for different n. With increasing values of n, the graph converges towards a Dirac delta function $\delta_{0.5}$, which has its peak at 0.5.



(b) Comparison between calculating the message length using the relative frequency and using the expectation value of the message length in the prior distribution for different values of a and b. With increasing values, the differences vanishes. The blue surface shows the message length value using the relative frequency; the green semi-transparent surface shows the message length value using the expectation value.

Figure 1.48: Using the Beta distribution as prior

Calculating the expectation value is very expensive. Thankfully, the difference between using the relative frequency and calculating the expectation value is negligible for bigger sample sizes as figure 1.48b shows.

In short, we can use our approximation in most cases without any difference. To adjust the frequencies, we simple have to adjust the numerator and denominator to the counted values:

⟨*Adjust frequency of a Bernoulli variable*⟩≡
```
const float adjustedFrequency = (numMatches + 1.0f) / (numSamples + 2.0f);
```

Then, we can use the vanilla definition of the message length to compute it:

⟨*Message length*⟩≡
```
float getMessageLength(float adjustedFrequency) {
  const float messageLength = -logf(safeProbability);
  return messageLength;
}
```

Validation for a Bernoulli sub-context

Table 1.49 shows different configurations for a single Bernoulli sub-context. Every two configurations are symmetric, so we group them as (a), (b), (c) and (d). We want to determine how much information each configuration conveys and verify if the message length fits our intuition:

(a) adds very little information since everything matches.

(b) tells us that the test message does not fit. Its value is unexpected since all example messages agree on the alternative outcome. This contains a lot of information.

	(a)		(b)		(c)		(d)	
test message	1	0	1	0	1	0	1	0
example message 1	1	0	0	1	1	0	0	1
⋮	1	0	0	1	1	0	0	1
	1	0	0	1	1	0	0	1
example message $n-1$	1	0	0	1	1	0	0	1
example message n	1	0	0	1	0	1	1	0

Table 1.49: Different sample configurations for a single Bernoulli sub-context. We want to determine the amount of information each configuration contains and what it can tell us.

(c) is, obviously, similar to (a). For test message n, we see that it does not fit the test message well since all other example messages together with the test message agree on the same result. So while little information has been gained about the other example messages, we have gained more information about example message n.

(d) shows that the test message does not match most of the example messages, which says little about the majority of the example messages, but it shows that example message n and the test message are probably more strongly correlated. Compared to (c), we can say this positive correlation is stronger than the negative correlation in the former.

How can we compute a message length for each configuration? We need to identify what our actual message is. If we look at the whole configuration, our message length is sum of the message lengths of all messages:

$$\sum_{k=1}^{n} h(E_k) + h(T), \tag{1.50}$$

where E_k is the k-th example message and T is the test message.

Usually, we only look at the message length of the test message and one of the example messages. Their combined message length is

$$h(E_k) + h(T). \tag{1.51}$$

This matches our expectations very well. For one, it fits our observation for (d): the match of the test message and example message n are assigned a higher message length than the mismatch of the test message with all other example messages. It also distinguishes between (c) and (d): (d) contains more information than (c) about how much the test message is correlated with message n.

All in all, we can conclude that this approach can be used to create importance weights.

Chapter 2

Algorithms

In this chapter we describe our contributions and how they work using a naive implementation.

2.1 Overview

Assisted Object Placement can be included in a designer's workflow without changing it too much.

When the level designers want suggestions for objects to place, they create a query volume in the level and perform a query. A **query volume** specifies the space for which to generate suggestions. This could be a sphere that is clipped against the level geometry (to limit the influence of space that the user does not see on the results), or it could be a simple oriented box. We use the latter for simplicity.

Assisted Object Placement performs the query and returns a list of **candidates** that suits the environment around the query volume. The candidates can be shown in a sidebar for easy selection, or it can be shown next to the query volume. See figure 2.2 on the next page for a comparison. The level designers can then scroll through the list and select the object they prefer and place it in the scene.

We use two different **contexts** to learn from a scene:

- a **probe context** that places probes in the scene and samples the environment of objects; and
- a **neighborhood context** that analyzes the neighborhood of objects using the distances between them.

In detail, this means that the probe context creates probes for all models that will be supported as candidates and uses them to sample the environment of all instances of these models in the scene. The samples reflect what the scene looks like near each instance. When the level designer starts a query, the query volume is sampled using probes as well, and we compare these against those of the different sampled models and rank the models according to how well their probe samples match those of the query volume.

Likewise, the neighborhood context uses the spatial relationship between objects. Specifically, it only uses the distances. Again, these distances are stored for all instances, and when a query for a query volume is performed, we determine the distances to all neighboring instances and try to find the models which have instances with a similar neighborhood.

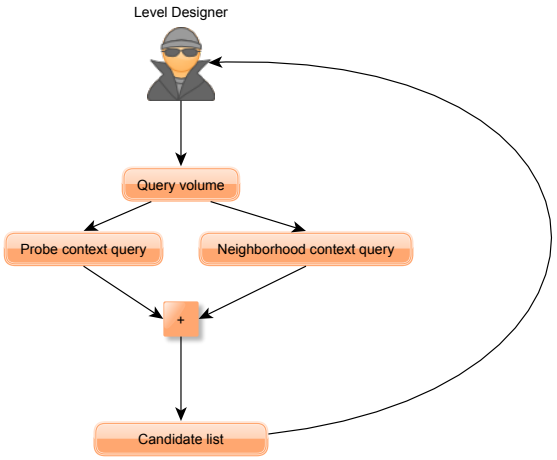


Figure 2.1: Workflow overview. The level designer creates a query volume which is used to perform queries on the probe and neighborhood databases. The results are combined to create a candidate list for the level designer.

(a) The canidate list is displayed next to the query volume

(b) The canidate list is displayed in a sidebar

Figure 2.2: Comparison of the two possible candidate bar positions. The query volume is selected and a query has been performed.

The results from both contexts are combined to form the final candidate list that is presented to the user. See figure 2.1 on the facing page for a diagram that visualizes the steps. We use the neighborhood context to refine the results from the probe context by purging false positives: false positives are models that are suggested because the geometry seems to fit, but that do not fit at all into the neighborhood of the query volume.

Assisted Object Placement keeps the sampled data in several databases, which can easily be updated in background to adapt to changes in the scene: as individual operations on a level are usually local to a small area, only data in those areas has to be resampled and updated, which could happen in a background thread.

We have examined different queries with varying performance and quality of the computed candidates. They differ in the way the information of the contexts is evaluated to create a score for each model.

2.2 Matching using probes

This section describes what the probe context is, how we sample the environment with probes, and how we match them to find candidates.

2.2.1 Motivation

Multiple ways exist to create a context based on an object's environment in a 3D scene: For example, we could create a context that samples the environment using spherical texture maps as done in [CWW11]. This works well for point entities that were examined in that research work, but it is not suited for our case because we want to take the geometry of the objects into account. To do this, we have to create multiple probes and distribute them over the model. But each probe samples the whole environment, so much of the probe samples will contain redundant data.

[NK03], [FH10] and [FSH11] use Zernike descriptors to describe the geometry of objects, but these descriptors cannot represent an open environment well.

Another approach has been used in [CLS10]: here, many light-weight probes that each only sample part of the environment are spread over the model. This has the benefit that we only have to work with conceptually simple entities and that we can combine the sampled information from different instances more easily. Moreover, we can create the probes depending on the geometry of the model without sampling too much redundant data. Once this is done, we do not have to care about a model's geometry any more.

We want to distinguish between probes, scene probes and probe samples. For each model, we create a number of **probes** that have different positions and directions depending on the model's geometry. They are stored in the model's coordinate system. For each instance of a model, we take the probes of its model and transform them to match the instance's **configuration**, that is its position and orientation, in the scene. We call these transformed probes **scene probes** because they belong in the scene. We sample the scene using these scene probes and store the resulting **probe samples** in a **probe database**. In brief, the probes are specific to a certain model and the probe samples to an instance of a model. Scene probes are the "glue" that connect probe samples to probes. See figure 2.3 on the next page for an illustration.

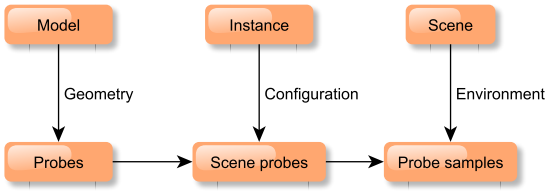


Figure 2.3: Relationship between models, instances and the scene on the one hand, and probes, instances and scenes probes on the other hand.

This separation allows us to correlate different probe samples from the same probe, and in effect treat each probe of a model as a probability distribution with its probe samples as example dataset. In the terms of section 1.5, the probes are the sub-contexts of the **probe context** of a model, and sampling the scene can be interpreted as computing the context map for the model.

2.2.2 Probe generation and probe samples

Probe generation

A probe has a position and a direction. It is positioned in a regular Cartesian grid, that might have been translated and rotated freely in space, but that must not have been scaled: the resolution of the grid stays fixed. Likewise, a probe's direction is chosen from a list of predefined directions. As a result, comparing probe samples from different models or query volumes is straight-forward. Moreover, the `Probe` structure can be kept very small:

⟨*Probe declaration*⟩≡

```
struct Probe {
  /*signed*/ char3 gridPosition;
  unsigned char directionIndex;
};
```

We can use integer coordinates to specify the position of a probe inside the grid. We place the origin of the grid at its center. Assisted Object Placement is intended for assisting with small objects, so using a `char3` for `gridPosition` is not a limitation. `directionIndex` specifies the direction. We support 26 different directions which point towards the 26 neighbors of a cell in a Cartesian grid:

⟨*neighborOffsets definition*⟩≡

```
const Vector3i neighborOffsets[26] = {
  // z
  Vector3i(0, 0, 1), Vector3i(0, 0, -1), Vector3i(-1, 0, -1),
  Vector3i(1, 0, -1), Vector3i(-1, 0, 1), Vector3i(1, 0, 1),
  Vector3i(0, 1, -1), Vector3i(0, -1, -1), Vector3i(-1, 1, -1),
  // x
  Vector3i(1, 0, 0), Vector3i(-1, 0, 0 ), Vector3i(1, 1, -1),
  Vector3i(1, -1, 1), Vector3i(1, 1, 1), Vector3i(1, -1, -1),
  Vector3i(-1, 1, 0), Vector3i(1, 1, 0), Vector3i(-1, -1, -1),
  // y
  Vector3i(0, 1, 0), Vector3i(0, -1, 0), Vector3i(-1, -1, 0), Vector3i(1, -1, 0),
  Vector3i(-1, -1, 1), Vector3i(0, -1, 1), Vector3i(-1, 1, 1), Vector3i(0, 1, 1)
```

```
};
```

⟨*directions definition*⟩≡
```
Vector3f directions[26];

void initDirections() {
  for(int directionIndex = 0; directionIndex < 26; directionIndex++) {
    directions[directionIndex] = neighborOffsets[i].cast<float>().normalized();
  }
}
```

The directions used here are important for determining the orientation of a candidate later. Other configurations are possible, but the one above is very simple and contains all the directions one would expect. See section 3.4 on page 51 for more details.

Generating probes for a query volume We use an oriented box as query volume. It can be defined by an isometric transformation and its size:

⟨*QueryVolume declaration*⟩≡
```
struct QueryVolume {
  Affine3f transformation;
  Vector3f size;
};
```

These properties together with the grid resolution already define the grid that we can use to place the probes. As we do not know where the candidates' probe samples will best fit in the grid, we sample all possible directions at each grid position:

⟨*generateQueryProbes*⟩≡
```
void generateQueryProbes(
  const Vector3f &size,
  const float resolution,
  Probes &probes
) {
  const Vector3i halfProbeExtent = ceil(size / (2.f * resolution) - Vector3f::Constant(0.5f));

  ⟨generateQueryProbes, range check⟩
  ⟨generateQueryProbes, reserve memory⟩

  Probe probe;
  for(int directionIndex = 0 ; directionIndex < boost::size( neighborOffsets ) ; directionIndex++) {
    for(
       auto gridIterator = Grid::makeIterator(-halfProbeExtent, halfProbeExtent)
     ; gridIterator.hasMore()
     ; gridIterator++
    ) {
      probe.position = gridIterator.toPosition();
      probe.directionIndex = directionIndex;
      probes.push_back(probe);
    }
  }
}
```

The `transformation` is not needed when we create the probes. We only need it later to transform the probes into the scene probes for the query volume. They are then used to sample the scene.

Generating probes for a model We create all probes for a specific model only once and store them in a **model database**. Generally, we do not want to create probes for all positions and directions; otherwise, a lot of redundant information would be sampled, which only slows down our algorithms.

We have examined several ways to generate probes for models. We can create probe positions by iterating over:

1. the full grid of the model's bounding box; or
2. non-empty voxels in a voxelization of the model.

Depending on the approach, we also have different options for choosing the directions for which we create probes at a certain position. If we use (1), we can use the position itself to restrict the set of directions. If we use (2), we can use the grid position, the averaged normals of the voxels, or neighborhood information. See section 3.3 on page 45 for more details on how we voxelize models and section 3.4 on page 51 for an evaluation of different options to select the probe directions.

Content of a probe sample

We can sample many properties, but we have chosen three that our tests have shown to be sufficient in most cases. Specifically, we sample: occlusion, distance, and color.

⟨*ProbeSample declaration*⟩≡

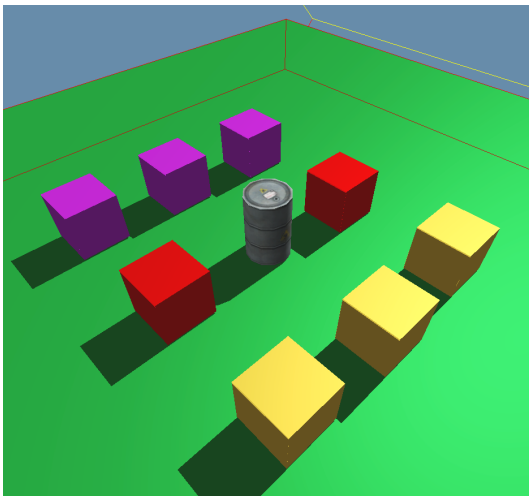
```
struct ProbeSample {
  /*signed*/ char3 colorLab;
  unsigned char occlusion;
  float distance;
};
```

We measure these quantities along a probe's direction up to a user-specified maximum distance. That is, we define a cone with an angle of 45° and trace rays randomly inside the cone starting from the probe's position up to a user-specified max distance. We use a cone of 45° so that we cover the whole environment at a probe's position when we sample all possible directions.

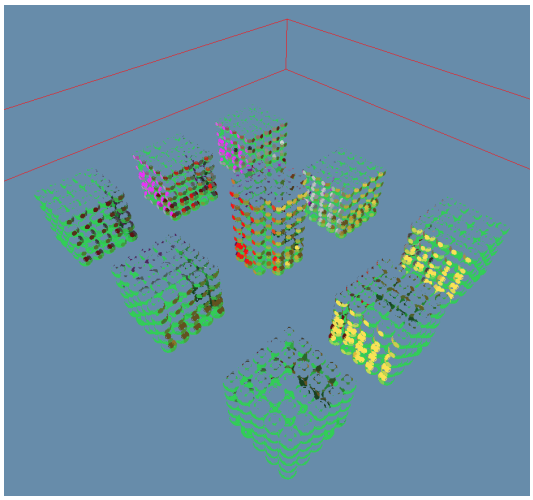
The ratio of rays that have hit something to the total number of rays approximates the occlusion inside the cone, and the average color and hit distance are used as a color and distance of the probe sample. This approach is more robust compared to tracing a single ray in the probe's direction because it is less prone to noise.

It is obvious that color and distance are useful properties, but what about occlusion? Why is it necessary? It is an integrated quantity and thus useful because our sampling resolution is low on the one hand; and on the other hand we sample many rays in a wide cone. Without occlusion, we could not tell if only one of the rays has hit something or all of them. Furthermore, without occlusion, we would have to assign a color to probe samples that do not hit anything. This would be bad because such a color can only be arbitrary since there are two possibilities when there is no hit: either no solid object is around, or the max distance of all rays is reached before they could hit a solid object. We could specify a sky color for the former, but there is no good "miss" color for the latter. Figure 2.4 on the next page shows a simple scene and visualizes the different properties of a probe sample.

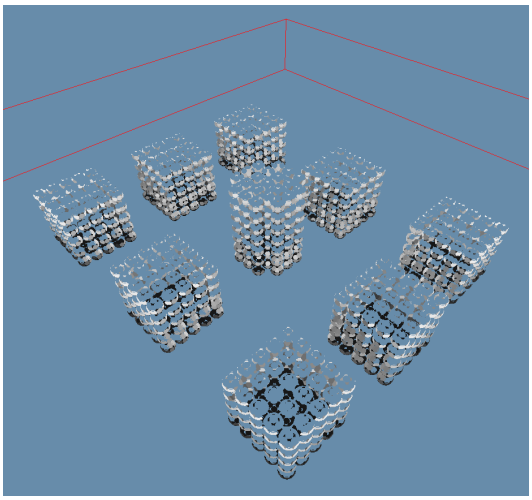
Sampling We sample the probes using NVIDIA's Optix raytracing framework. We have found that sampling each probe with 127 rays offers a good trade-off between speed and quality in our



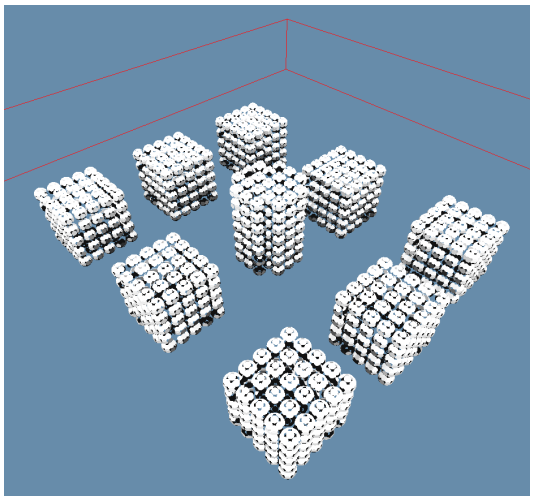
(a) Scene



(b) Sampled colors



(c) Sampled distances. The darker, the nearer; light blue is the "miss" color.



(d) Sampled occlusion. The brighter, the less occluded.

Figure 2.4: Visualization of probe samples in a simple scene. All models except for the green floor are sampled. The small disks represent the probe samples. They are drawn in the sky color if there was no hit, that is when there is zero occlusion.

test cases. The ray directions are chosen from a pre-generated set of random cone directions. The distribution is cosine-weighted. Thus it chooses ray directions that point right into the probe's direction more often.

Matching probe samples

Two probe samples match if all their properties match within certain tolerances. We compare distances and occlusion using their absolute difference to see if they match.

Colors are converted to the **CIELAB** color space, because its difference metric is the Euclidean metric, which can be efficiently calculated. It offers better results than comparing colors in RGB because it attempts to model human visual perception. Its metric is normed in that way that a "**just noticable difference**" of colors has a distance of about 1.0. [MvEO94] has found a different value of about 2.3 through a survey. Good sources for information about color models and colormetry are [Sto03] and [AMHH08]. [Khe02] and [MvEO94] provide many details about CIELAB and color difference metrics.

⟨*Probe matching*⟩≡

```
bool matchColor(const ProbeSample &a, const ProbeSample &b, const float squaredTolerance) {
  return (a.colorLab - b.colorLab).squaredNorm() <= squaredTolerance;
}

bool matchDistance(const ProbeSample &a, const ProbeSample &b, const float tolerance) {
  return fabs(a.distance - b.distance) <= tolerance;
}

bool matchOcclusion(const ProbeSample &a, const ProbeSample &b, const int integerTolerance) {
  return abs(a.occlusion - b.occlusion) <= integerTolerance);
}

bool match(const ProbeSample &a, const ProbeSample &b, ProbeContextTolerance &probeContextTolerance) {
  return
     matchOcclusion(a, b, probeContextTolerance.occlusion_integerTolerance)
   &&
     matchDistance(a, b, probeContextTolerance.distance_tolerance)
   &&
     matchColor(a, b, probeContextTolerance.colorLab_squaredTolerance)
  ;
}
```

The distance tolerance has to be at least half the resolution of the grid. Otherwise, shifting the grid could lead to unwanted mismatches.

2.2.3 Finding candidates

We sample the query volume and use one of various queries to find the best candidates. A query returns a list of candidates. Each candidate has a score, usually between 0 and 1, that is an estimate for how well it matches the query volume. This corresponds to the probability of a model for a given probe context as it was defined in section 1.5 on page 4.

The simplest queries only calculate this score, while more elaborate queries can also suggest a placement position and orientation. We have implemented:

- bidirectional match queries,

- configuration queries, and
- bidirectional importance-weighted match queries.

Each query iterates over all sampled models and matches the probe samples of a model against the ones of the query volume to calculate a score for each model. This score can be computed independently. The calculations can thus be parallelized efficiently. Once all scores have been computed, the models can be ranked by their score. The scores can be normalized to make their sum equal 1, or they can be interpreted as maximum-likelihood values. We have found the latter to be less confusing in the user interface.

Bidirectional match query

The **bidirectional match query** is the simplest query. For each sampled model, it calculates the ratio of the probe samples in the query volume that match a probe sample in the model to all probe samples in the query volume and the ratio of probe samples in the model that match a probe sample in the query volume to all probe samples in the model. Then it multiplies both ratios to find an estimate for the similarity of the two probe contexts. See [CLS10] for the rational behind it.

It can be implemented by keeping track for each probe sample whether it has been matched or not and then by counting all matches.

⟨*Bidirectional match query*⟩≡

```
float BidirectionalMatchQuery::matchAgainst(int modelIndex) {
  const auto &sampledModel = probeDatabase.sampledModels[modelIndex];
  const ProbeSamples &modelProbeSamples = sampledModel.getProbeSamples();

  vector<bool> matchedQuerySamples(queryProbeSamples.size());
  vector<bool> matchedModelSamples(modelProbeSamples.size());
  for(int querySampleIndex = 0 ; querySampleIndex < queryProbeSamples.size() ; ++querySampleIndex) {
    for(int modelSampleIndex = 0 ; modelSampleIndex < modelProbeSamples.size() ; ++modelSampleIndex) {
      if(match(
        queryProbeSamples[querySampleIndex],
        modelProbeSamples[modelSampleIndex],
        probeContextTolerance
      )) {
        matchedQuerySamples[querySampleIndex] = true;
        matchedModelSamples[modelSampleIndex] = true;
      }
    }
  }

  const float matchedQuerySamplesRatio = float(matchedQuerySamples.count()) /
    queryProbeSamples.size();
  const float matchedModelSamplesRatio = float(matchedModelSamples.count()) /
    modelProbeSamples.size();

  const float score = matchedModelSamplesRatio * matchedQuerySamplesRatio;
  return score;
}
```

This query already yields good results in practice even though it ignores the original direction of a probe. An explanation for this is that there is already a lot of variance in the direction of

many small props: they are rotated to point in different directions or turned on the side; so a probe's direction carries little information.

Configuration query

For better results, we use a **configuration query**: it determines the best configurations for each candidate. The **configuration** of an instance consists of its position and orientation. Only positions inside the query volume are allowed, and we also only support a fixed number of orientations: we support all orientations that exactly rotate each probe direction onto another probe direction. This makes it easy to write efficient algorithms for it: we can invert the problem and ask which probe directions are mapped onto each other by a specific orientation and compare those probe samples with each other to create a score for that orientation. The only orientations that can do this for our 26 available probe directions are the ones that are obtained by rotations around the main axes in 90° steps. This gives us 24 possible orientations (that preserve handedness). See section 3.4 on page 51 for more details.

⟨*Configuration query*⟩≡
```
ConfigurationResult ConfigurationQuery::matchAgainst(int modelIndex) {
  const auto &sampledModel = probeDatabase.sampledModels[modelIndex];
  const ProbeSamples &modelProbeSamples = sampledModel.getProbeSamples();
  const Probe &modelProbes = sampledModel.getProbes();

  ⟨Configuration query, create score buffer 24⟩
  ⟨Configuration query, compute scores 24⟩
  ⟨Configuration query, return result 25⟩
}
```

We store separate scores for all possible configurations of a model in the query volume.

⟨*Configuration query, create score buffer*⟩≡
```
vector<GridBuffer<int>> configurationScores(
  ProbeGenerator::numOrientations,
  GridBuffer<int>(queryVolume.size)
);
```

When two probe samples match, we use both probes' direction and position to determine where an instance would have to be placed for these probes' direction and position to match and increment the scores for these configurations.

⟨*Configuration query, compute scores*⟩≡
```
for(int querySampleIndex = 0 ; querySampleIndex < queryProbeSamples.size() ; ++querySampleIndex) {
  for(int modelSampleIndex = 0 ; modelSampleIndex < modelProbeSamples.size() ; ++modelSampleIndex) {
    if(match(
      queryProbeSamples[querySampleIndex],
      modelProbeSamples[modelSampleIndex],
      probeContextTolerance
    )) {
      ⟨Configuration query, handle match 24⟩
    }
  }
}
```

⟨*Configuration query, handle match*⟩≡
```
const Probe &modelProbe = modelProbes[sampledModel.getProbeIndex(modelSampleIndex)];
```

```
const Probe &queryProbe = queryProbes[querySampleIndex];

const vector<int> &possibleOrientations = ProbeGenerator::getMatchingOrientations(
  modelProbe.directionIndex,
  queryProbe.directionIndex
);

for(auto orientation = possibleOrientations.begin()
  ; orientation != possibleOrientations.end()
  ; orientation++
) {
  const auto relativePosition = ProbeGenerator::transformVectorByOrientation(
    modelProbe.position,
    *orientation
  );

  const auto targetPosition = queryProbe.position - relativePosition;
  configurationScores[*orientation].tryAdd(targetPosition, 1);
}
```

In other words, we calculate a likelihood field for the different configurations in the query volume.

We normalize this field by dividing by the number of probes of the model to get a measure that is independent of it. A value of 0 means that no probe sample has been matched, while a value of 1 means that as many probes have been matched as the model contains. Then, the likelihood fields of different models can be compared. To compare the results to the ones of the bidirectional match query, we select the highest-scoring configuration and return its score for each model.

⟨*Configuration query, return result*⟩≡

```
ConfigurationResult configurationResult;
⟨Configuration query, select best candidate⟩

configurationResult.score /= modelProbes.size();
return configurationResult;
```

We also support returning all candidate configurations. The best instances and their configurations can then be suggested to the level designer independently of the model.

Importance-weighted queries

The **bidirectional importance-weighted match query** is based on the bidirectional match query but enhances it by weighting each probe sample by how important it is. For this, we use the importance weight we examined in section 1.5.3 on page 11. We use the color of a probe sample to estimate its importance. To do this, we quantize the colors and calculate the color frequencies over all probe samples.

⟨*Bidirectional importance-weighted match query*⟩≡

```
float BidirectionalMatchQuery::matchAgainst(int modelIndex) {
  [···like ⟨Bidirectional match query 23⟩···]
  const float matchedQuerySamplesRatio = weightedRatio(matchedQuerySamples, queryProbeSamples);
  const float matchedModelSamplesRatio = weightedRatio(matchedModelSamples, modelProbeSamples);

  const float score = matchedProbeSamplesRatio * matchedQuerySamplesRatio;
  return score;
}
```

`weightedRatio` calculates the weighted ratio using two helper functions `ProbeDatabase::getHitImportance` and `ProbeDatabase::getMissImportance`:

⟨*weightedRatio*⟩≡
```
float weightedRatio(const vector<bool> &matchedSamples, const ProbeSamples &probeSamples) {
  float totalSum = 0.0f;
  float totalWeightSum = 0.0f;

  for(int sampleIndex = 0 ; sampleIndex < matchedSamples.size() ; ++sampleIndex) {
    if(matchedSamples[sampleIndex]) {
      const float sampleImportance = probeDatabase.getHitImportance(probeSamples[sampleIndex]);
      totalSum += sampleImportance;
      totalWeightSum += sampleImportance;
    }
    else {
      const float sampleImportance = probeDatabase.getMissImportance(probeSamples[sampleIndex]);
      // totalSum += 0.0f;
      totalWeightSum += sampleImportance;
    }
  }

  return totalSum / totalWeightSum;
}
```

`ProbeDatabase::getMissImportance` is slightly more complex than `ProbeDatabase::getHitImportance` because we have to calculate the probability of the complement:

⟨*ProbeDatabase::getMissImportance*⟩≡
```
float ProbeDatabase::getMissImportance(const ProbeSample &sample) {
  const float adjustedFrequency = getAdjustedFrequency(sample);
  const float informationEntropy = -logf(1.0f - adjustedFrequency);
  return informationEntropy;
}
```

⟨*ProbeDatabase::getHitImportance*⟩≡
```
float ProbeDatabase::getHitImportance(const ProbeSample &sample) {
  const float adjustedFrequency = getAdjustedFrequency(sample);
  const float informationEntropy = -logf(adjustedFrequency);
  return informationEntropy;
}
```

`ProbeDatabase::getAdjustedFrequency` returns the adjusted frequency of the probe sample's color. Adjusted frequencies are described in section 1.5.3 on page 11.

2.3 Matching using neighborhood distances

2.3.1 Motivation

The probe context is not sufficient for more complex scenes because it does not incorporate any kind of semantic information. For one, it is possible to have models that always appear in groups and look similar to another group of models. In the most extreme case, models could share the same appearance and only differ semantically by using a different game script. There is no way for the probe context to differentiate between them. Furthermore, the probe context is limited by its resolution and maximum distance, and most importantly, it prefers false positives over false

negatives because it calculates scores per sampled model and not per sampled instance. This is good in general because it makes viable suggestions even when we only have few probe samples. On the other hand, it will suggest a model for environments that partially match separate instances of that model even if no single sampled instance would have matched the whole environment well.

We want to filter out illogical suggestions. We do this by looking at one kind of semantic information that is readily available in every game: we look at the actual objects in the scene and the distances between them.

2.3.2 Distance sets

Idea

To determine whether a model is a good candidate for a query volume, we look at all of its instances and the neighbors they have and compare them to the neighbors of the query volume. Specifically, we only look at distances of an instance to all its neighbors. The inherent rotation invariance of this simplifies matching.

Representation

All instances of a model are identical, so there is no need to distinguish between neighbors of the same model type. Only the number of instances in a certain distance is important.

As a result, we can represent the neighborhood context as a map of distance multi-sets: for each neighbor model, we keep a multi-set of distances to all its instances up to a certain maximum distance $d_{\max}$. A **multi-set** is like a set except that it can contain the same element multiple times. Nonetheless, we will use the shorter term distance set instead of distance multi-set and acknowledge that the same distance can appear more than once in it.

See figure 2.5 on the following page for an illustration.

Probability model

Context map We can apply the notation from the probability model described in section 1.5 on page 4. We set our scene coordinate space $\mathbb{U} = \mathbb{R}^3$ and use a neighbor context map C_N that is defined as follows:

$$\mathsf{C}_N \colon \mathbb{U} \longrightarrow \mathbb{M} \longrightarrow D \subset \{\!\{d \in [0, d_{\max}]\}\!\}\,. \tag{2.6}$$

$\{\!\{d \in [0, d_{\max}]\}\!\}$ stands for a multi-set with elements in the interval $[0, d_{\max}]$. To simplify the notation, we again use this variant:

$$\mathsf{C}_N \colon \mathbb{U} \times \mathbb{M} \longrightarrow D \subset \{\!\{d \in [0, d_{\max}]\}\!\} \tag{2.7}$$

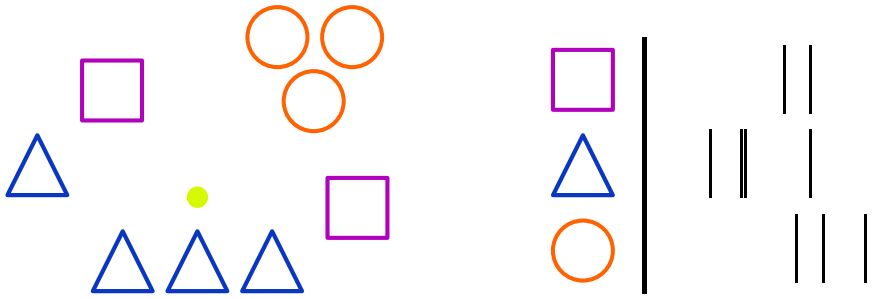
The neighbor context-map maps a position in the scene and a model to a set of distances.

The distances in figure 2.5 on the following page could be written as follows:

$$\mathsf{C}_N\Big(\bullet, \square \Big) = \{\!\{32, 38\}\!\} \tag{2.8}$$

$$\mathsf{C}_N\Big(\bullet, \triangle \Big) = \{\!\{15, 23, 23, 38\}\!\} \tag{2.9}$$

$$\mathsf{C}_N\Big(\bullet, \bigcirc \Big) = \{\!\{35, 42, 52\}\!\} \tag{2.10}$$



(a) View of the example scene. The green dot symbolizes the position at which the context is sampled.

(b) Neighbor context of the position. The bars signify the distance to one neighbor instance of the respective model. The two bars that almost touch represent a distance that appears twice in the multi-set.

Figure 2.5: Example neighborhood

Problem formulation and sub-contexts Here, we do not calculate $P\big(\mathcal{M} \,\big|\, \mathsf{C}_N(\mathbf{p})\big)$, instead we calculate $P\big(\mathbf{I} \,\big|\, \mathsf{C}_N(\mathbf{p})\big)$ directly for all instances. We then rank the models by their best instance. This keeps the neighborhood context from behaving like the probe context by separating the contexts of different instances and forces matches to be stricter.

We can rewrite $P\big(\mathbf{I} \,\big|\, \mathsf{C}_N(\mathbf{p})\big)$ using the definition of conditional probability as

$$P\big(\mathbf{I} \,\big|\, \mathsf{C}_N(\mathbf{p})\big) \propto P(\mathbf{I}, \mathsf{C}_N(\mathbf{p})). \tag{2.11}$$

To find $P(\mathbf{I}, \mathsf{C}_N(\mathbf{p}))$, we split the context into one sub-context for each possible neighbor model and apply section 1.5.2 on page 8 to write

$$P(\mathbf{I}, \mathsf{C}_N(\mathbf{p})) = E_{\hat{\Phi}}\left[P(\mathsf{C}_N(\mathbf{p}, \mathcal{N}), \mathbf{I})\right], \tag{2.12}$$

with a normalized importance weight $\hat{\Phi}$. We will later investigate different choices for this importance weight. To sum it up, we quantify for each neighbor model how well it matches the respective sub-context and use an importance weight to determine $P(\mathbf{I}, \mathsf{C}_N(\mathbf{p}))$.

To calculate the probability, we need to determine how good the instance distance sets match the query distance sets. We can split this into two parts:

1. we need to group distances that match; and
2. we need to create a score that quantifies how good a distance set matches the query distance set.

This score will represent the probability of the sub-context up to normalization.

Comparing distance sets

First, let us compare only two distance sets. We do this by trying to find distances in both sets that match. When do two distances from different distance sets match? Until now, we have not

•	0	1	0	1	0	0
□$_1$	0	0	0	1	1	0
□$_2$	1	1	1	0	1	0
○	0	0	0	1	1	1

Table 2.14: Representation of grouped distances in the algorithm using the distances from figure 2.13. The columns are color coded to visualize the group they belong to in figure 2.13c.

looked at one issue: the distances are distributed very sparsly. Two entries in distance sets will rarely have the same value, so we need to match distances within a tolerance. We use the diagonal length of the model for this. This means that two instances (in different distance sets) that would overlap in space are matched as well.

We do not want to find candidates for a specific point in space: we have a query volume, and we try to find candidates for it. Because of this, we use an additional tolerance value, called the **query tolerance**. Its value is the diagonal length of the query volume. This ensures that we could choose any point inside the query volume to calculate the distances without changing the results of the algorithm.

We can establish that two distances match if they are equal up to these tolerances. A distance in one set can be matched at most once because each distance represents one instance in the neighborhood, and we want to match instances one-to-one to distinguish between differently sized groups of objects. We call a distance that could not be matched to a distance in the query distance an **unmatched distance**.

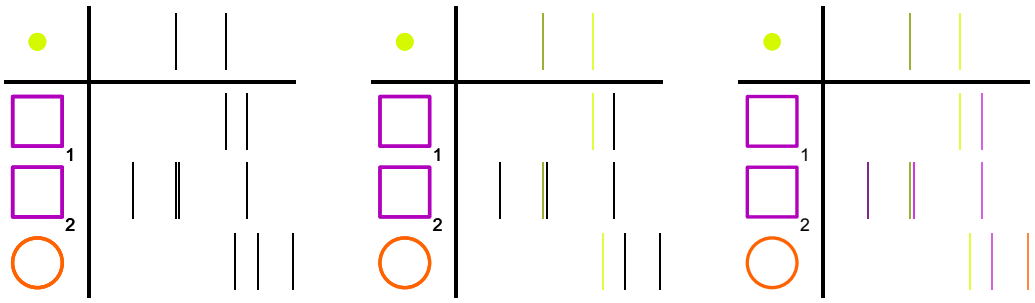
Therefore, if we compare two distance sets, we can iterate through the distance sets of each neighbor model using an algorithm similar to a merge sort and register matched and unmatched distances.

When we have multiple instance distance sets that we want to compare with a query distance set, we do the same: we start with the query distances and try to find matching distances in each distance set. Again, each query distance can match at most one distance in a distance set. Note though that the same query distance can match distances in multiple instance distance sets at the same time. See figure 2.13a on the following page and figure 2.13b for an illustration.

All unmatched distances can now be matched among themselves to find groups of distances that refer to the “same” neighbor that has not been found in the query distance set. Figure 2.13c illustrates this. The grouping of unmatched distances allows us to identify missing instances from different instance distance sets and gives us additional information regarding how unexpected unmatched distances are. This can be used to create an importance weight using section 1.5.3 on page 11.

When we are done with this, we have many groups of distances from different distance sets that either match a query distance or do not. We can represent this using a binary table. Each column represents a group of distances, and each row represents a different distance set (including the query distance set). An entry is 1 or 0 depending on whether the specific distance set has a distance that belongs to the group or not. See table 2.14 for an illustration.

The problem of finding a similarity measure for neighborhood contexts can thus be mapped to



(a) Query distance set and instance distance sets before any distances have been grouped

(b) Query distance set and instance distance sets after matching distances have been grouped

(c) Query distance set and instance distance sets after unmatched distances have been grouped as well

Figure 2.13: Matching a query distance set with a neighborhood database. The distance sets are shown for an neighbor model that is not shown here. The neighborhood database contains two instances of □ and one instance of ○.

measuring the similarity of bit strings. We have looked at two well-known metrics for comparing two bit strings:

- the Rand measure; and
- the Jaccard index.

The **Rand measure** uses the simplest approach to compute the similarity between two bit strings $x = (x_1, \ldots, x_n)$ and $y = (y_1, \ldots, y_n)$ by calculating the ratio $R(x, y)$ of matching bits to all bits:

$$R(x, y) = \frac{m_{11} + m_{00}}{m_{11} + m_{10} + m_{01} + m_{00}}, \tag{2.15}$$

with

$$m_{ij} = \left|\left\{k \;\middle|\; x_k = i \wedge y_k = j\right\}\right|. \tag{2.16}$$

It was introduced in [Ran71].

There is one issue with using the Rand measure: with a growing instance count in the database and a growing number of sampled neighbors, the number of m_{00} will grow since the probability increases that distance are left unmatched, and the Rand measure ratios will tend towards 1 as the amount of this noise increases. This approach does not seem robust. However, we have to keep in mind that, even though the values move towards 1, the ordering of the instances stays correct. Instances with more overlap with the query have a higher score.

Nonetheless, this does not match our expectations: the more noise we have, the lower our calculated similarity measure should be.

The **Jaccard index** $J(x, y)$ solves this problem by ignoring m_{00}:

$$J(x, y) = \frac{m_{11}}{m_{11} + m_{10} + m_{01}}. \tag{2.17}$$

For some thoughts on the probabilistic basis of the Jaccard index, see [RV96]. The Jaccard index has been used successfully in recent research papers for pattern matching, see [NC08], [NSMO10] and [SAMO11].

Our case is slightly different, however: we do not want to compare two bit strings, we have many bit strings that we want to compare at the same time with one query bit string. Additional information is available and we can use it.

We have come up with a similarity measure that uses section 1.5.3 on page 11 and message lengths to create an importance weight. Specifically, we calculate an adjusted frequency for each column and use the combined message length formula from equation (1.51) on page 14 to weight it:

$$\Phi\big(k, l \;\big|\; \mathsf{C}_N, \mathbf{I}_\mathbf{l}\big) = -\log\left(\frac{\#\{d_q\}_k + 1}{\#\{0,1\}_k + 2}\right) - \log\left(\frac{\#\{d_l\}_k + 1}{\#\{0,1\}_k + 2}\right), \tag{2.18}$$

where d_q is the query bit string, d_l is the bit string belonging to instance $\mathbf{I}_l$ and $\#\{x\}_k$ is the number of bits with value x in the k-th column of the table. We call this similarity measure the **importance-weighted (similarity) measure**.

The Jaccard index could be interpreted as using a weight function which weighs a 00-match with 0 and all other matches with 1. This interpretation leads us to the question whether we could create a weight function that uses message lengths but assigns $0,0$ matches a weight of 0.

2.3.3 Implementation

The actual algorithm we have implemented consists of three analogous parts:

- group distances in the neighborhood context database with distances in the query context;
- group unmatched distances with themselves; and
- calculate a score for each instance in the database.

As suggested by the probability model, the algorithm only deals with one neighbor model at a time. It returns scores for all sampled instances for every neighbor model. The scores are merged afterwards. We start by grouping instance distances with query distances. All unmatched distances are grouped in a second step. The implementation separates scoring from matching even though the two are interleaved during execution. It uses a special class `Scores` to encapsulate the scoring modalities.

⟨*matchAgainstNeighborModel*⟩≡

```
Scores matchAgainstNeighborModel(Id sceneNeighborModelId) {
  const int totalNumInstances = database.getTotalNumInstances();

  Scores neighborModelCandidateScores(totalNumInstances);

  vector<UnmatchedDistance> unmatchedDistances = matchDistances(
    sceneNeighborModelId,
    neighborModelCandidateScores
  );

  processUnmatchedDistances(
    sceneNeighborModelId,
    move(unmatchedDistances),
    neighborModelCandidateScores
  );

  return neighborModelCandidateScores;
}
```

A special class `Scores::MatchedDistances` is used to save information about matched distances. We iterate over all models and instances and group matched distances before we integrate the matches into the scores.

⟨*matchDistances*⟩≡
```
UnmatchedDistances matchDistances(
  Id sceneNeighborModelId,
  Scores &neighborModelCandidateScores
) {
  ⟨matchDistances, fetch constants 32⟩

  UnmatchedDistances unmatchedDistances;
  Scores::MatchedDistances matchedDistances(totalNumInstances, numQueryDistances);

  ⟨matchDistances, iterate over models and instances 32⟩

  neighborModelCandidateScores.integrateMatchedDistances(matchedDistances);
  return unmatchedDistances;
}
```

We initialize the algorithm and determine the model-specific tolerance value.

⟨*matchDistances, fetch constants*⟩≡
```
const int numSampledModels = database.sampledModelsById.size();
const int totalNumInstances = database.getTotalNumInstances();

const Distances &queryDistances = queryDataset.getDistances(sceneNeighborModelId);
const int numQueryDistances = queryDistances.size();

const float neighborModelTolerance = getNeighborModelTolerance(sceneNeighborModelId);
```

⟨*getNeighborModelTolerance*⟩≡
```
float getNeighborModelTolerance(Id id) {
  return modelDatabase->informationById[id].diagonalLength * 0.5f;
}
```

To be flexible in how we calculate the probabilities later, we want to keep the distance groups as finely grained as possible: for each instance of each model, we remember whether it matches a certain query distance or not. We keep a `globalInstanceIndex` which uniquely identifies every instance of every model in our database while we iterate over all instances.

⟨*matchDistances, iterate over models and instances*⟩≡
```
int globalInstanceIndex = 0;
for(int candidateModelIndex = 0 ; candidateModelIndex < numSampledModels ; ++candidateModelIndex) {
  const SampledModel &candidateModel = database.sampledModelsById[ candidateModelIndex ].second;

  for(int instanceIndex = 0 ; instanceIndex < candidateModel.instances.size() ; ++instanceIndex,
    ++globalInstanceIndex) {
    const auto &instance = candidateModel.instances[ instanceIndex ];
    const Distances &instanceDistances = instance.getDistances(sceneNeighborModelId);

    ⟨matchDistances, group instance distances and query distances 33⟩
  }
}
```

The algorithm iterates over all query distances while it either pushes unmatched instance distances into the `unmatchedDistances` container or remembers the match for later.

⟨*matchDistances, group instance distances and query distances*⟩≡
```
auto instanceDistance = instanceDistances.begin();
for(int queryDistanceIndex = 0 ; queryDistanceIndex < numQueryDistances ; ++queryDistanceIndex) {
  ⟨matchDistances, compute the query distance interval 33⟩
  ⟨matchDistances, process unmatchable distances 33⟩
  ⟨matchDistances, match distance 33⟩
}

⟨matchDistances, process remaining unmatchable distances 34⟩
```

We scale `neighborModelTolerance` linearly depending on the distance when we calculate the distance interval for each query distance. This ensures that neighbor instances that are further away have more leeway. This makes the algorithm cope better with higher values for `maxDistance`.

⟨*matchDistances, compute the query distance interval*⟩≡
```
const float queryDistance = queryDistances[ queryDistanceIndex ];
const float queryDistanceToleranceScale = getDistanceToleranceScale(queryDistance);

const float tolerance = queryDistanceToleranceScale * neighborModelTolerance + queryTolerance;
const float beginQueryDistanceInterval = queryDistance - tolerance;
const float endQueryDistanceInterval = queryDistance + tolerance;
```

We start by processing instance distances that cannot be matched by a query distance because they lie outside any query distance interval. Unmatchable instance distances are stored in a special structure that stores the distance, the model index and the global instance index.

⟨*matchDistances, process unmatchable distances*⟩≡
```
while(
   instanceDistance != instanceDistances.end()
  &&
   *instanceDistance < beginQueryDistanceInterval
) {
  unmatchedDistances.emplace_back(UnmatchedDistance(*instanceDistance, candidateModelIndex,
    globalInstanceIndex));

  ++instanceDistance;
}
```

⟨*UnmatchedDistance declaration*⟩≡
```
struct UnmatchedDistance {
  float distance;
  int candidateModelIndex;
  int globalInstanceIndex;

  UnmatchedDistance(float distance, int candidateModelIndex, int globalInstanceIndex)
    : distance(distance)
    , candidateModelIndex(candidateModelIndex)
    , globalInstanceIndex(globalInstanceIndex)
  {}

  ⟨UnmatchedDistance static methods⟩
};
typedef vector<UnmatchedDistance> UnmatchedDistances;
```

Then, we either match the current instance distance if possible, or interrupt the loop if there are no more distances for the current instance.

⟨*matchDistances, match distance*⟩≡

```
if(instanceDistance == instanceDistances.end()) {
  break;
}

if(*instanceDistance <= endQueryDistanceInterval) {
  matchedDistances.match(globalInstanceIndex, queryDistanceIndex);

  ++instanceDistance;
}
```

Only one instance distance can be matched, so, after matching it, the loop continues with the next query distance. Finally, when all query distances have been processed, the remaining instance distances are marked as unmatched.

⟨*matchDistances, process remaining unmatchable distances*⟩≡

```
for(; instanceDistance != instanceDistances.end() ; ++instanceDistance) {
  unmatchedDistances.emplace_back(UnmatchedDistance(*instanceDistance, candidateModelIndex,
    globalInstanceIndex));
}
```

Now that we have dealt with all distances in the database that can be grouped together with a query distance, we process the unmatched distances. We try to group them with each other, so that we get a list of unmatched distances that represent the "same" neighbor instance that has not been found in the query's neighborhood.

The algorithm takes the container of unmatched distances as parameter, groups them and updates the scores.

⟨*processUnmatchedDistances*⟩≡

```
void processUnmatchedDistances(
  Id sceneNeighborModelId,
  UnmatchedDistances &&unmatchedDistances,
  Scores &neighborModelCandidateScores
) {
  const int totalNumInstances = database.getTotalNumInstances();
  const float neighborModelTolerance = getNeighborModelTolerance(sceneNeighborModelId);

  ⟨processUnmatchedDistances, process unmatched distances 34⟩
}
```

We process the unmatched distances iteratively because we cannot deal with all of them at once. Instead, the algorithm tries to process as many as possible and defers the ones it cannot process to the next iteration. In every iteration at least one unmatched distance is processed, so it will terminate. In general, many more unmatched distances are processed in each iteration.

⟨*processUnmatchedDistances, process unmatched distances*⟩≡

```
UnmatchedDistances deferredUnmatchedDistances;
deferredUnmatchedDistances.reserve(unmatchedDistances.size());

while(!unmatchedDistances.empty()) {
  ⟨processUnmatchedDistances, process a batch of unmatched distances 35⟩

  unmatchedDistances.swap(deferredUnmatchedDistances);
  deferredUnmatchedDistances.clear();
}
```

The sequence of unmatched distances needs to be sorted by distance, so that the algorithm can easily collect all distances inside a range. We could use a `stable_sort` here as it is already partially

sorted: `matchDistances` stores unmatched distances implicitly sorted by global instance index and distance—in this order. Unmatched distances are then processed in bins by:

1. collecting all unmatched distances inside a distance interval specified by the tolerance values;
2. only accepting one unmatched distance per instance;
3. deferring all other unmatched distances to the next batch;
4. updating the scores; and
5. repeating this until no more unmatched distances are left in the current batch.

⟨*UnmatchedDistance public methods*⟩+≡

```
static bool less_by_distance(const UnmatchedDistance &a, const UnmatchedDistance &b) {
 return a.distance < b.distance;
}
```

⟨*processUnmatchedDistances, process a batch of unmatched distances*⟩≡

```
boost::sort(unmatchedDistances, UnmatchedDistance::less_by_distance);

auto unmatchedDistance = unmatchedDistances.begin();
while(unmatchedDistance != unmatchedDistances.end()) {
 ⟨processUnmatchedDistances, collect unmatched distances inside a tolerance interval 35⟩
 ⟨processUnmatchedDistances, accept one unmatched distance per instance and defer the rest 36⟩

 neighborModelCandidateScores.integrateGroupedUnmatchedDistances(move(groupedUnmatchedDistances));
}
```

To collect unmatched distances, we set up a distance interval starting at the first unprocessed unmatched distance and iterate over all unmatched distances that lie in the distance interval.

⟨*processUnmatchedDistances, collect unmatched distances inside a tolerance interval*⟩≡

```
const auto binBegin = unmatchedDistance;
{
 const float beginDistance = binBegin->distance;
 const float toleranceScale = getDistanceToleranceScale(binBegin->distance);

 const float endDistance =
    beginDistance
  +
   2 * neighborModelTolerance * toleranceScale
  +
   2 * queryTolerance
 ;

 do {
  ++unmatchedDistance;
 } while(
    unmatchedDistance != unmatchedDistances.end()
  &&
    unmatchedDistance->distance <= endDistance
 );
}
const auto binEnd = unmatchedDistance;
```

After this, we sort this sub-range by the global instance index and iterate over it again. We only keep the first unmatched distance for each candidate and defer all other unmatched distances of the instance to the next iteration.

```
⟨UnmatchedDistance public methods⟩+≡
 static bool less_by_globalInstanceIndex(const UnmatchedDistance &a, const UnmatchedDistance &b) {
  return a.globalInstanceIndex < b.globalInstanceIndex;
 }

⟨processUnmatchedDistances, accept one unmatched distance per instance and defer the rest⟩≡
 sort(binBegin, binEnd, UnmatchedDistance::less_by_globalInstanceIndex);

 vector<int> groupedUnmatchedDistances;
 for(auto binElement = binBegin ; binElement != binEnd ;) {
  const int globalInstanceIndex = binElement->globalInstanceIndex;
  groupedUnmatchedDistances.push_back(globalInstanceIndex);
  ++binElement;

  while(
     binElement != binEnd
   &&
     binElement->globalInstanceIndex == globalInstanceIndex
  ) {
   deferredUnmatchedDistances.push_back(*binElement);
   ++binElement;
  }
 }
```

Calculating the candidate scores

The scores are then calculated using one of the three similarity measures we have introduced. Different versions of `Scores::integrateMatchedDistances` and `Scores::integrateGroupedUnmatchedDistances` are used depending on the measure we use. These are straight-forward implementations of the formulas we have seen in section 2.3.2.

2.4 Combined matching

The probe context only samples the environment of an object locally, and the neighborhood context samples it globally. However, it does not sample the immediate surroundings of an object very well.

Both contexts sample the environment independently of each other except for a small overlap for which their data is interconnected by the geometry of the models. Thus, we can assume that they are almost independent variables, and we can approximatively combine them as such:

$$P(\mathsf{C}) = P(\mathsf{C_N}, \mathsf{C_P}) = P(\mathsf{C_N})\, P(\mathsf{C_P}). \tag{2.19}$$

This works well in practice; see section 4.2.3 on page 74 for results.

Chapter 3

Implementation

3.1 Demonstration application

Our demonstration application shows that Assisted Object Placement works and allows us to test it. With it, we can:

- load artificial test scenes and levels from Shadowgrounds Survivor;
- place new instances;
- sample models into the probe and neighborhood databases;
- create query volumes;
- manipulate instances and query volumes;
- perform queries and examine the created candidate lists; and
- automatically place a candidate.

In addition, we can use it to examine various internal aspects of Assisted Object Placement. We can visualize:

- models, their voxelizations and the generated probes in the model database;
- probe samples for all instances and query volumes; and
- likelihood fields that are generated by configuration queries,

and we can control various settings for the databases and queries, and dump everything for off-line validation tests.

Most of the screenshots in this thesis were created using it. Figures 3.1 and 3.2 on the next page and on page 39 show our demonstration application and its user interface.

Frameworks and libraries

We have used the following frameworks and libraries:

- **Boost** to avoid reimplementing many algorithms;
- **Eigen** for linear algebra and vector operations;
- **OpenMP** and **Microsoft's Parallel Pattern Library** for parallelizing;
- NVIDIA's **Optix** and **CUDA** for sampling instance probes;
- **RectangleBinPack** for merging textures into a single big texture (for Optix);
- **OpenGL** and **GLEW** for real-time rendering;

Figure 3.1: Screenshot of the demonstration application. It shows the level `marine02_road`.

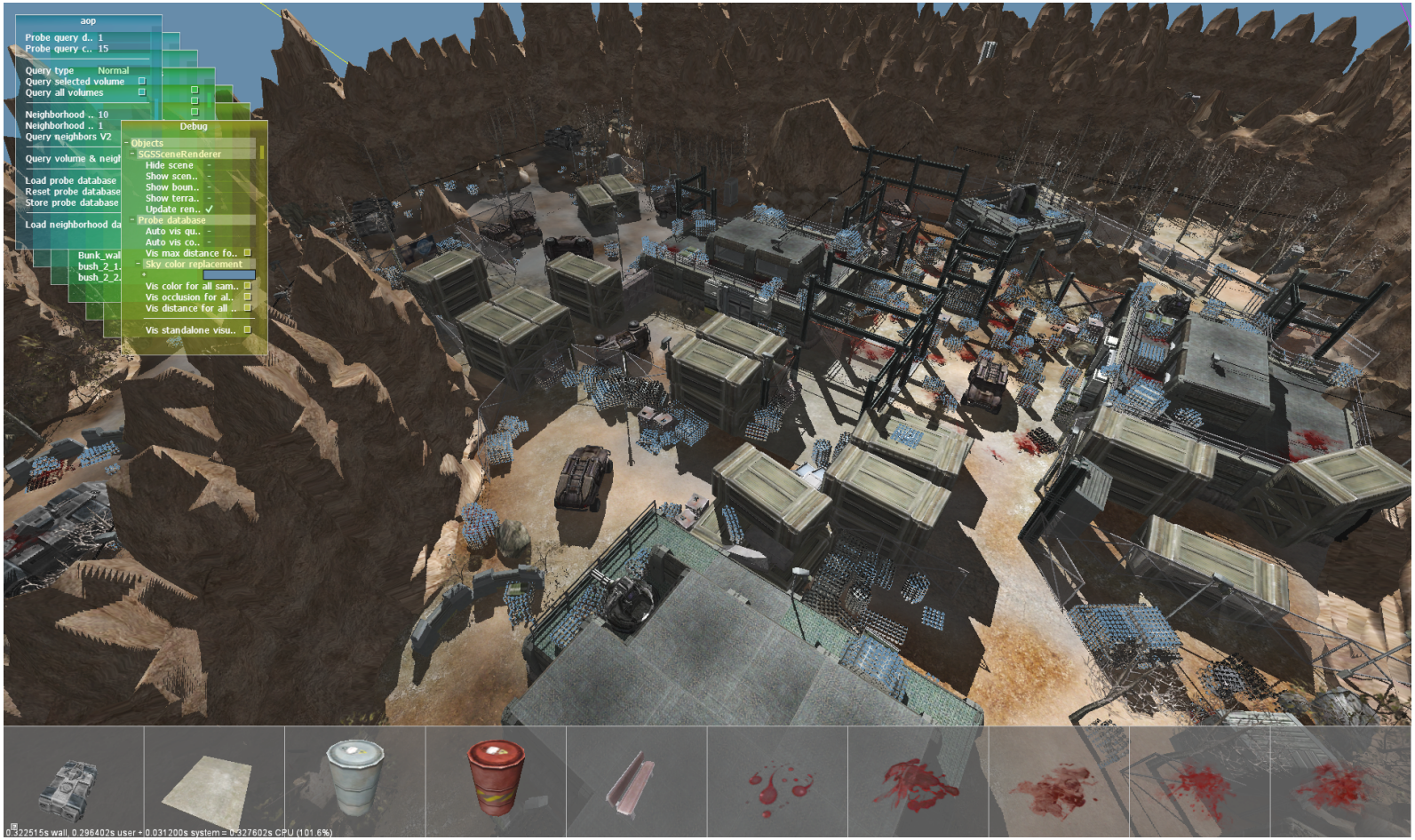


Figure 3.2: Screenshot of the demonstration application. It shows the level `marine02_road`. All probe samples in the database have been visualized in the scene for debugging purposes.

- **SFML** for creating windows and handling input;
- **SOIL** for loading images;
- **AntTweakBar** for providing a simple user interface; and
- **GoogleTest** for writing unit-tests.

Additionally to using these libraries, the code consists of 25000 *LoC*.

3.2 Interaction with Shadowgrounds Survivor

The asset pipeline is an often underestimated but very important part of any project dealing with visualization. In the following section we are going to describe briefly how we export level and model data from Shadowgrounds Survivor and how we load and render it in the demonstration application.

3.2.1 Asset pipeline

We have specifically chosen a game that has been open sourced so that we did not have to reverse engineer any file formats or tools. Shadowgrounds Survivor's source code has been released in 2011. The released source code has been uploaded to a Mercurial repository hosted by Google Projects[1]. Accessing data from the game is possible by:

- using its model viewer;
- loading a level in the editor; or
- loading a level in the game.

For Assisted Object Placement, it is important to look at existing levels and to examine how objects are placed. Consequently, we have decided to look at the level editor in more detail and opted to extend it to export levels into other file formats. The advantages of it are that we can:

- navigate and examine different levels easily;
- edit levels when needed;
- use the augmented level data of the editor instead of the raw level files; and
- extend an existing user interface.

See figure 3.3 on the next page for a screenshot of the level editor. On the other hand, the level editor's code base is a lot bigger than that of the model viewer, and it takes several steps to export a level because it requires UI interactions.

Since the editor is a big code base with many files, we determined to separate our code from the original code as much as possible: by concentrating our code in one location, we can avoid costly changes later. For this, we have used the visitor pattern, which is described in [Gam95]: First, we created an interface with all necessary callback functions to accept models, materials and instances, and then we implemented a small `visitGameObjects` method in all important classes of the editor. Finally, we added an Export to.. entry to the File menu, which

1. opens a save dialog;
2. creates the corresponding exporter vistor; and at last
3. calls the root `visitGameObjects` method with the visitor.

[1] http://code.google.com/p/shadogrounds-and-shadowgrounds-survivor/

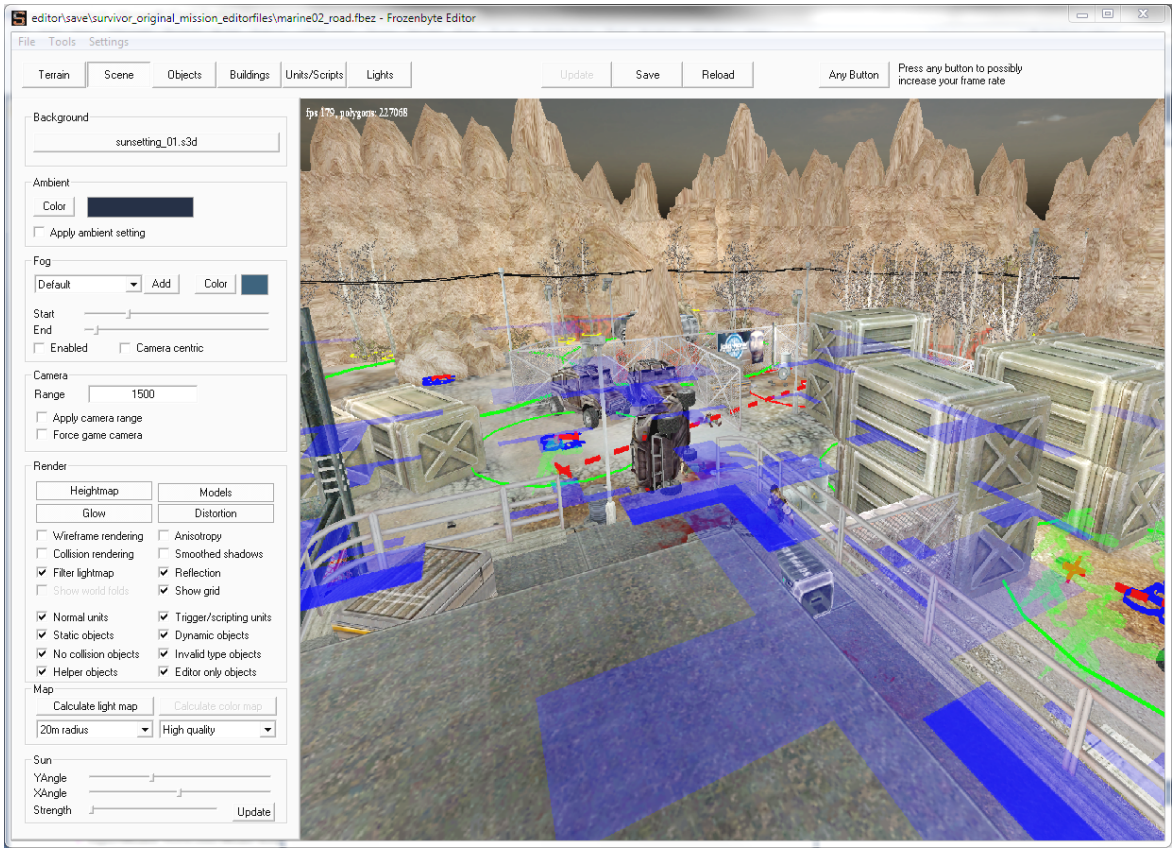


Figure 3.3: A screenshot of Shadowgrounds Survivor's level editor. It shows the level `marine02_road`.

Except for the initial addition of the `visitGameObjects` method, we did not have to change anything in the editor's code afterwards.

3.2.2 Model and level data

A level in Shadowgrounds Survivor consists of three parts:

- terrain;
- buildings that have a separate floor model; and
- objects.

The terrain is stored using a height-map and supports multiple texture layers. Each texture layer has its own weight texture. The weight textures are used to blend the detail textures of the different layers.

Buildings and regular game objects are treated the same internally: they are composed of sub-models, which consist of mesh data and a material definition. A building's floor model includes

special sub-models. When the player enters a building, the regular sub-models are hidden and the floor sub-models are shown in their place.

A material consists of two base textures, which can be blended using different operations, and additional special-effect textures for reflections, bump mapping and distortions. Additionally, it specifies properties like diffuse and specular color and the alpha type of the sub-model, which specifies whether it is transparent or not. There are different alpha types. A sub-model can be opaque or it can:

- use a material-specific transparency value;
- use its texture's alpha channel;
- use additive transparency; or
- use multiplicative transparency.

Our demonstration application fully supports the terrain. It ignores a building's floor model, as well as other collision models and editor-only objects. It only supports one texture, the diffuse material color, and the various alpha types. This is sufficient for rendering almost all scenes correctly.

3.2.3 Exporters

We have implemented three exporters:

- a `.csv` exporter for object positions;
- an `.obj` exporter for geometry data; and
- an `.sgsScene` exporter.

`.sgsScene` is our own format.

`.csv` exporter

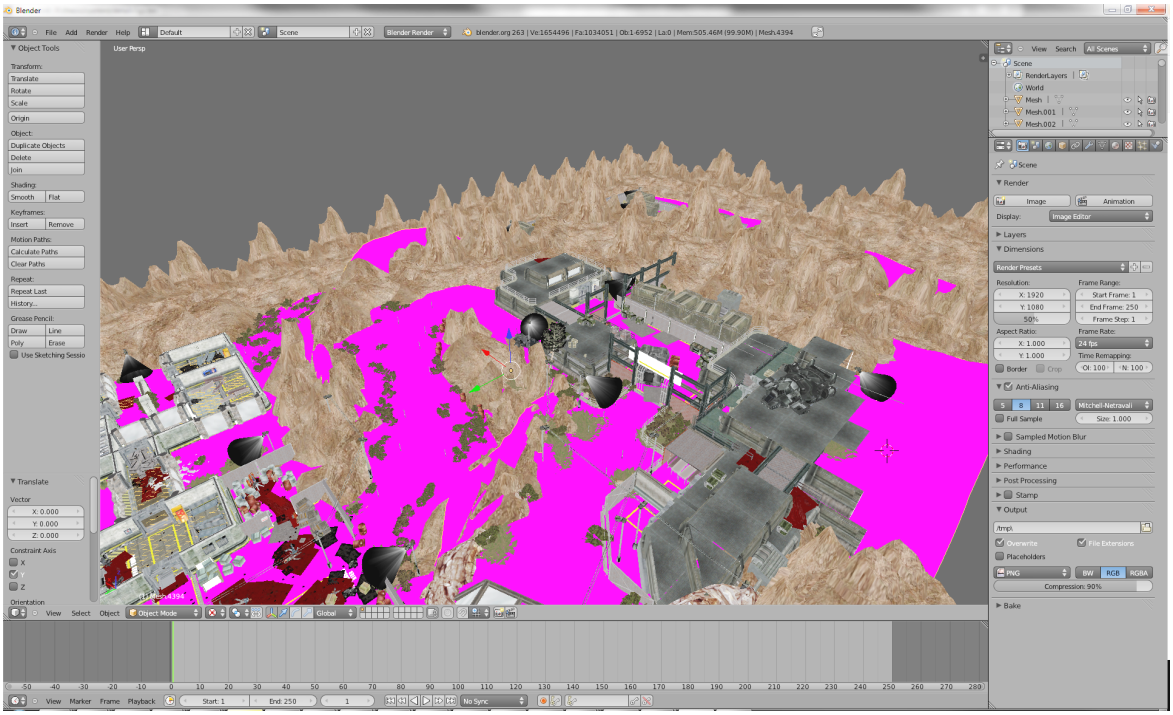
The `.csv` exporter writes information about an object like its position, orientation and bounding box into a `.csv` file. This was only used during the initial orientation phase of our research to calculate probability densities of objects depending on their distances to each other.

`.obj` exporter

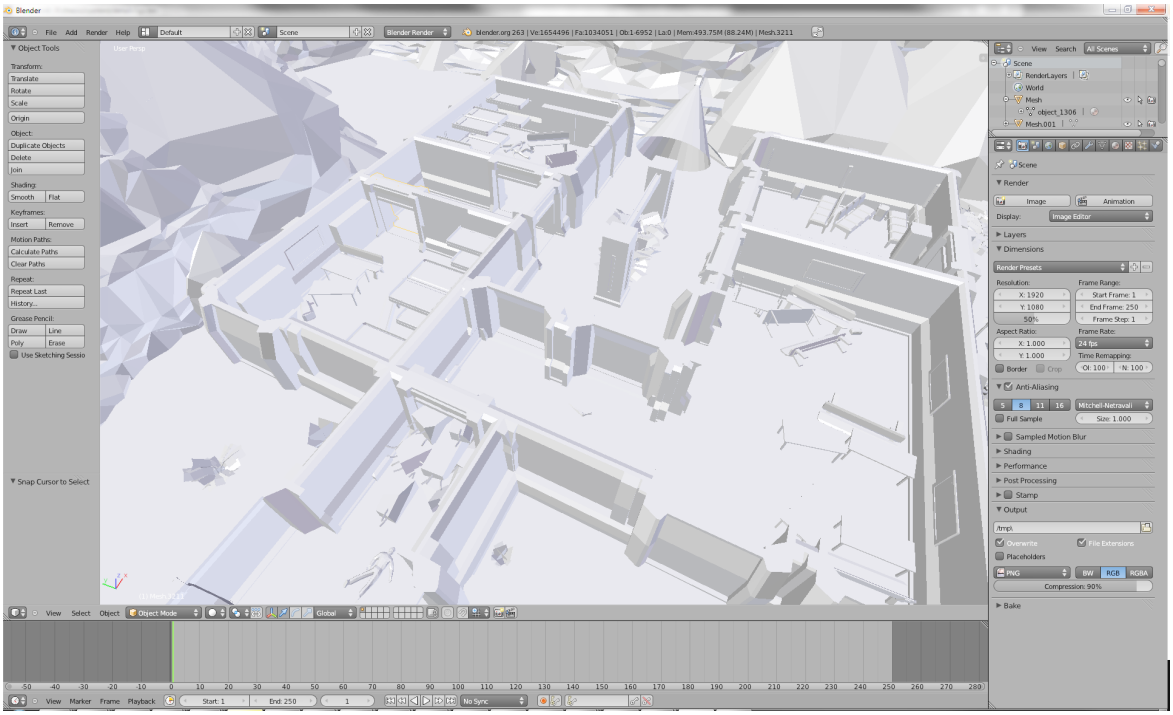
The `.obj` exporter can export all models and sub-models into an `.obj` file. It also supports exporting the terrain without textures. This exporter was intended as the preferred way to import geometry, but there were several problems:

- the loading and saving an `.obj` file takes very long;
- the file format is not standardized any more: different applications interpret material properties differently;
- the material file does not support arbitrary texture file names; and, most importantly,
- the format is not suited for storing levels that contain multiple instances of the same sub-model because the model geometry has to be replicated.

Using the `.obj` exporter, exporting the level `marine01_wakeup` from Shadowgrounds Survivor takes minutes and creates an 180 MB big `.obj` file. See figure 3.4 on the facing page for two screenshots of the `.obj` version of this level.



(a) with textures



(b) without any materials

Figure 3.4: `.obj` version of the Shadowgrounds Survivor level `marine01_wakeup` loaded in Blender

.sgsScene exporter

For these reasons, we have decided to write a custom binary format that specifically suits the needs of our demonstration application. The format is called .sgsScene., and it can store the level marine01_wakeup and all related data in just 54 MB.

In addition, it is designed to directly dump its data structures to disk, and it is self-contained, which means that it contains all textures and models needed to display a level of Shadowgrounds Survivor. This makes loading and displaying a level very easy.

Format details To avoid processing any textures in the exporter, we store all needed texture files as raw byte buffers in the .sgsScene file.

For the terrain, we store all texture layers together with their weight textures to be able to process them later. We do not store the terrain's heightmap. Instead, we create a terrain mesh in the exporter and store it as tiles of 8×8 vertices that can be separately culled when they lie outside the viewing frustum. As it is, the terrain could contain any solid mesh.

For each model, we calculate its bounding box and bounding sphere, and also export all its sub-models. They consist of a mesh and material definition as well as their own bounding box and bounding sphere. The bounding box and bounding sphere are important for dynamically culling objects during rendering.

Last but not least, all game objects in the level are exported. Only their configuration (position and orientation) needs to be stored since Assisted Object Placement does not require any semantic information.

Serializer To simplify saving and loading an .sgsScene file, we have created a simple serializer. It provides helper macros to serialize data structures:

```
struct Texture {
  string name;
  vector<unsigned char> rawContent;

  SERIALIZER_DEFAULT_IMPL((name)(rawContent));
};

struct Color4ub {
  unsigned char r, g, b, a;

  SERIALIZER_ENABLE_RAW_MODE();
};
```

This enables us to quickly adapt the file format during code iterations. Everything is contained in one header file, and it can be shared easily between different projects like the level editor and our demonstration application.

Using such an implicit approach is dangerous, however, when switching between 32-bit and 64-bit architectures. Specifically, it is important to ensure that types are correctly mapped. In Visual Studio, this is not a problem in general because the size of all usual integral types does not change. Special care has to be taken when serializing variables that represent pointer differences or container sizes: types like size_t have a different size depending on the architecture.

3.2.4 Rendering

We use vertex buffers to render the terrain and all game objects because NVIDIA's Optix needs all data packed into buffers as well. We use immediate mode and display lists to render everything else: we use display lists to apply different materials, and they are used for all debug visualizations because they work like a black-box after they have been compiled. We can render anything into them and later display it without worrying about what we have to display. This is very useful since providing an integrated debug interface becomes very simple with display lists. Current drivers are still optimized for display lists because 3D modeling applications still use them, even though they have been deprecated in OpenGL.

When the `.sgsScene` file is loaded, the terrain layers are blended together using their weight textures and baked into a single 8192^2 or 16384^2 texture. Therefore, unlike in the original game, which requires several shaders and blending of multiple textures, the terrain can be treated just like everything else and rendered by binding a single texture. This is particularly helpful as the Optix renderer would have to duplicate the shader code from the OpenGL renderer.

At the time of writing, Optix has limited texture support, and using different textures requires costly material program switches. For this reason, we merge all model textures into one big texture on load and modify the texture coordinates in the Optix material program on the fly.

The baked terrain texture and the merged model textures are cached in a different file to avoid recreating it every time the application starts. We cache the acceleration structure Optix creates in the same way.

3.3 Voxelizer

As mentioned in section 2.2,we voxelize models to determine good probe positions. We also want to calculate averaged normals to optionally determines probe directions with them. we have followed the approach described in [Cozl2, chapter 22] and also implemented conservative voxelization using the second algorithm explained in [PF05, chapter 42]. We use geometry shaders, ARB_shader_image_load_store (which is included in OpenGL 4.2), and atomic operations to directly render into a volume texture.

3.3.1 Basic implementation

We create a grid for a model's bounding box similar to the grids that are used for generating probe positions for query volumes. A volume texture is created for this grid, and the model is splatted into it using special shaders.

This is done by rendering the model into three different viewports using a geometry shader. Each viewport uses a different orthogonal projection: one for each main axis. Each triangle is assigned to the viewport that maximizes its area or, in other words, the number of generated fragments. This is necessary because triangles whose face normal is almost perpendicular to the view direction of an orthogonal view create too few fragments during rasterization which leads to undersampling and missing voxels, ie gaps in the voxelized surface of the model. See figure 3.5 on the next page for an example of this problem.

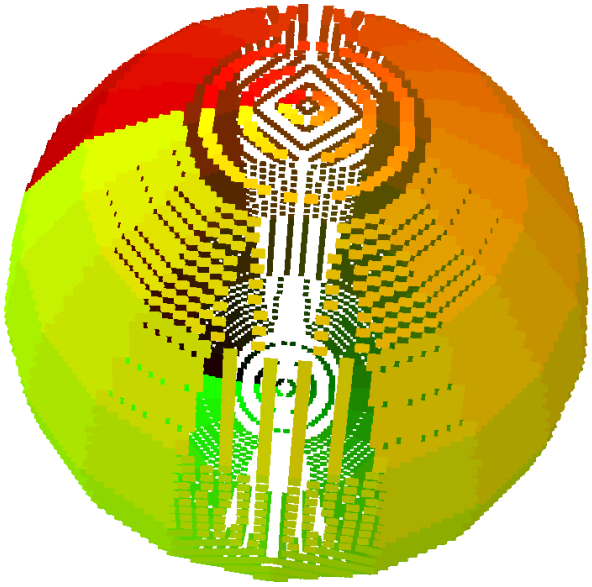


Figure 3.5: Voxelization of a sphere from single viewport. Faces that are almost perpendicular to the view direction suffer from severe undersampling.

The shader consequently chooses the viewport whose view direction maximizes the dot product with the triangle's face normal. Since the view directions are the unit vectors in $\mathbb{R}^3$, this boils down to finding the maximum absolute component of the face normal vector.

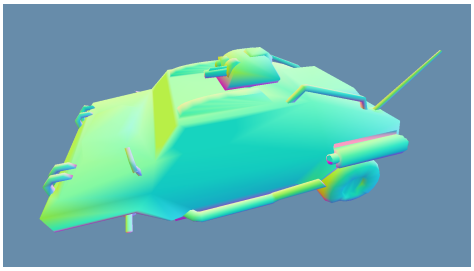
If conservative rasterization is not used, we simply render the triangle into the viewport. We enable `GL_RASTERIZATION_DISCARD` as we do not actually render anything into a conventional framebuffer. Instead, we use ARB_shader_image_load_store to splat directly into the volume texture[2].

Because multiple shader cores could attempt to access the same voxel in the volume texture, we use atomic operations to update it. We are interested in knowing whether a voxel is empty, and if not, what the average normal direction is. An RGBA8 volume texture would be perfect for storing the normal direction in its RGB components and information on whether the voxel is empty in its A component. [Cozl2] uses a running average calculation to avoid overflow issues with RGBA8. However, this results in unnecessarily complicated code and requires a `imageAtomicCompSwap` loop to avoid race conditions. Furthermore, OpenGL 4.2 supports atomic addition operations only for R32UI textures.

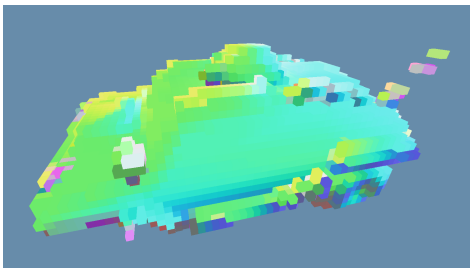
As a result, we use four separate R32UI volume textures: one for each channel. This has the advantage that overflow issues cannot occur and we can simple add up all normal components during voxel splatting and calculate the average as final step. For this, the A component is incremented every time a voxel is written. It represents the hit count for the voxel. The resulting average normal is certainly of higher quality than the normals calculated with a running average. Splatting one voxel only requires four calls to `imageAtomicAdd`.

[2] ARB_framebuffer_no_attachments could be used to avoid creating dummy framebuffers, but it was not supported on our workstations at the time of writing.

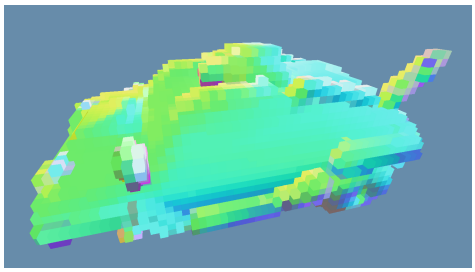
(a) Textured model



(b) Normals of the model



(c) Standard voxelization. Notice how small features like the car's antenna suffer from undersampling and disappear from the voxelization.



(d) Voxelization with conservative rasterization. Conservative rasterization ensures that each triangle spawns at least one voxel, capturing even small features correctly.

Figure 3.6: Voxelization of a model from Shadowgrounds Survivor

The disadvantage of this solution is the higher memory consumption: it also uses four times as much memory. On the other hand, we only voxelize models, so it is not a problem. Additionally, we need a final pass to calculate the average normals and to combine the four channels back into one RGBA8 volume texture. This could be done on the CPU, but we do it on the GPU as well: We set up a vertex array with a single element and perform one instanced draw call with one instance for each voxel in the volume texture. In the shader, we again use ARB_shader_image_load_store to read and write the volume textures.

3.3.2 Conservative rasterization

Conservative rasterization ensures that fragments are generated for all pixels that are covered by a triangle. Normally, fragments are only generated for pixels whose center is covered. Compare figure 3.6c with figure 3.6d: the antenna of the car is not correctly voxelized without conservative rasterization and suffers from undersampling.

The algorithm works by expanding each triangle by half a pixel. Thus, it ensures that, if a triangle covers only part of a pixel, the expanded triangle certainly covers the pixel center, too.

It expands a triangle by expanding its corners. For narrow triangles, this generates unnecessary fragments. It can be fixed by computing a bounding rectangle: the algorithm expands the bounding rectangle of the original triangle by half a pixel and later discards fragments in the fragment shader that lie outside of it. Figure 3.7 illustrates the idea.

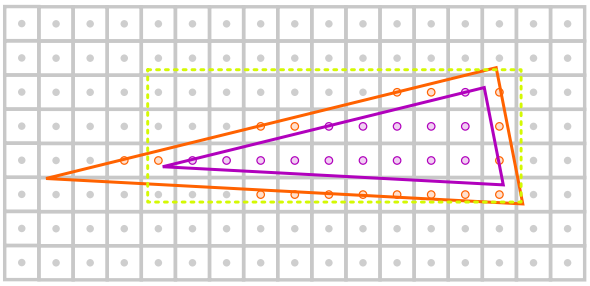


Figure 3.7: Triangle expansion by the conservative rasterizer. The original triangle is drawn in purple; the expanded triangle is drawn in orange; and the bounding rectangle is drawn in green. The purple fragments are generated by the original triangle; the orange fragments are the additional fragments generated by the expanded triangle. Notice that the expanded triangle generates one unnecessary fragment for the left corner, which is clipped by the bounding rectangle.

[PF05] employs optimized calculations that use projective geometry. It is concise and fast but not easy to follow and verify. Therefore we have implemented the algorithm ourselves using simpler regular vector geometry in $\mathbb{R}^2$, which we are going to describe now:

⟨*conservative rasterization algorithm*⟩≡
```
⟨set halfPixelSize 48⟩
⟨determine the bounding rectangle⟩

⟨expand the bounding rectangle 48⟩

⟨calculate screen edges 48⟩

⟨calculate outward normals 49⟩
⟨calculate edge shifts 49⟩

⟨expand triangle corners 51⟩
```

We begin by choosing the right coordinate system. We are not going to expand the triangles in clip-space coordinates. We are using screen coordinates because then we can simply set:

⟨*set halfPixelSize*⟩≡
```
const vec2 halfPixelSize = vec2(0.5);
```

We determine the bounding rectangle for the triangle and expand it by half a pixel.

⟨*expand the bounding rectangle*⟩≡
```
boundingBox.min -= halfPixelSize;
boundingBox.max += halfPixelSize;
```

We are going to expand the triangle in screen coordinates first, that is in 2D, and then we are going to correct the `z` components. For this, we need to calculate the edges of the triangle in screen space in counter-clockwise orientation.

⟨*calculate screen edges*⟩≡
```
const vec2 screenEdges[3] = {
  corners[1].xy - corners[0].xy,
  corners[2].xy - corners[1].xy,
  corners[0].xy - corners[2].xy
```

```
};
```

To expand the edges outwards, we need to know the outward normal for each edge, which we can determine using the edge and the face normal.

⟨*getOutwardNormal*⟩≡
```
vec2 getOutwardNormal(const vec2 edge, const vec3 faceNormal) {
  return normalize(cross(vec3(edge, 0.0), faceNormal).xy);
}
```

⟨*calculate outward normals*⟩≡
```
const vec2 screenOutwardNormals[3] = {
  getOutwardNormal(screenEdges[0], faceNormal)),
  getOutwardNormal(screenEdges[1], faceNormal)),
  getOutwardNormal(screenEdges[2], faceNormal))
};
```

Note that the face normal will always point into the screen or out of the it; otherwise, another viewport would have been chosen for the current triangle. This ensures that the cross product will have non-zero `xy` components.

To calculate the amount by which each edge is shifted, we apply a trick: we assume that the outward normal points into the first quadrant and that the origin lies on the edge. Then, we shift the edge so that the center of the pixel at the origin, that is `halfPixelSize`, lies on the shifted edge. This means that we shift the edge by `halfPixelSize`. Obviously, the edge need not lie on the origin for this to be true. If the outward normal does not point into the first quadrant but into another one, we flip the signs of the components of `halfPixelSize` so that it points into the same quadrant.

We will see shortly that it simplifies the calculations if the edge shift is perpendicular to the edge. This is equivalent to projecting the edge shift onto the outward normal. The outward normal is a unit vector, so we can implement this as

```
edgeShift = screenOutwardNormal * dot(screenOutwardNormal, halfPixelSize *
  sign(screenOutwardNormal)).
```
(3.8)

We can rewrite this as

```
edgeShift = screenOutwardNormal * dot(sign(screenOutwardNormal) * screenOutwardNormal,
  halfPixelSize),
```
(3.9)

which leads us to the final form:

```
edgeShift = screenOutwardNormal * dot(abs(screenOutwardNormal), halfPixelSize).
```
(3.10)

In the end we have:

⟨*calculate edge shifts*⟩≡
```
const vec2 edgeShift[3] = {
  screenOutwardNormals[0] * dot(abs(screenOutwardNormals[0]), halfPixelSize),
  screenOutwardNormals[1] * dot(abs(screenOutwardNormals[1]), halfPixelSize),
  screenOutwardNormals[2] * dot(abs(screenOutwardNormals[2]), halfPixelSize)
};
```

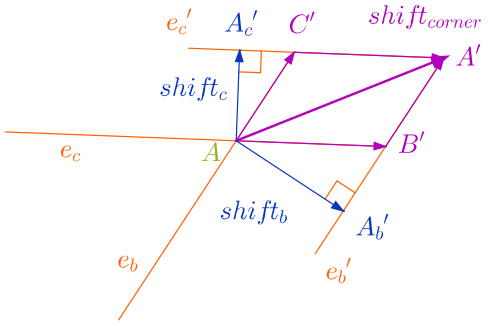


Figure 3.11: Geometric construction of $shift_{corner}$. A is the corner expand; e_b and e_c are the edges of the original triangle; $shift_c$ and $shift_b$ are the perpendicular edge shift vectors; C' and A' are the intersection points of one edge with the shifted version of the other edge; A' is the expanded triangle corner; $shift_{corner}$ is the shift vector for A.

Having calculated all this, we can now compute the shift for each corner of the triangle. Figure 3.11 shows the idea: We need to determine AC' and AB' to obtain $\overrightarrow{AA'} = \overrightarrow{AC'} + \overrightarrow{AB'}$. Since $C'A_c'A$ and $B'A_b'A'$ are right triangles, we can calculate AC' and AB' immediately.

⟨*unproject*⟩≡
```
// return λ*direction, such that λ*direction orthogonally projected onto v = v
vec2 unproject(const vec2 direction, const vec2 v) {
  return direction * dot(v, v) / dot(direction, v);
}
```

⟨*getCornerShift*⟩≡
```
vec3 getCornerShift(
  const vec3 faceNormal,
  const vec3 edgeB,
  const vec3 edgeC,
  const vec3 shiftEdgeB,
  const vec3 shiftEdgeC
) {
  vec3 cornerShift;
  cornerShift.xy = unproject(edgeB, shiftEdgeC) + unproject(edgeC, shiftEdgeB);

  ⟨determine cornerShift.z 50⟩

  return cornerShift;
}
```

We only need to find the necessary shift for the z component of the corner now. Because the shifted corner lies on the same plane in space as the triangle, we have

```
dot(faceNormal, corner) = dot(faceNormal, corner + cornerShift),
```
(3.12)

and we see that `dot(faceNormal, cornerShift)` has to be zero. We already know `cornerShift.xy`, so we can solve for `cornerShift.z` and are done.

⟨*determine cornerShift.z*⟩≡
```
cornerShift.z = -dot(faceNormal.xy, xyShift) / faceNormal.z;
```

```
⟨expand triangle corners⟩≡
 corners[0] += getCornerShift(
  corners[0], faceNormal,
  screenEdges[2], screenEdges[0],
  edgeShift[2], edgeShift[0]
 );
 corners[1] += getCornerShift(
  corners[1], faceNormal,
  screenEdges[0], screenEdges[1],
  edgeShift[0], edgeShift[1]
 );
 corners[2] += getCornerShift(
  corners[2], faceNormal,
  screenEdges[1], screenEdges[2],
  edgeShift[1], edgeShift[2]
 );
```

3.4 Probe directions

In section 2.2 we have noted that we have implemented multiple ways to determine which probe directions generate as little redundant data as possible and that the available directions are important for determining the number of orientations the configuration query can support. Now we are going to examine both topics.

3.4.1 Possible direction sets

The choice of the direction set is important for configuration queries as has been explained in section 2.2.3 on page 24: The rotation belonging to each orientation has to rotate the directions onto themselves. Otherwise, there is no exact inverse map, and our algorithm will not work. The questions that arise are: given a set of directions, what are the orientations that fulfill this requirement? And, how can we choose our direction set to maximize the number of such orientations? Finally, we can state an additional constraint: A level designer expects simple orientations to be suggested instead of skewed one, so orientations along the three main axes should be included.

How many such orienations exist for the 26 directions described in section 2.2.2 on page 18? The 26 directions can be imagined as being the 26 vertices of a **deltoidal icositetrahedron**. See figure 3.13 on the following page for an illustration. Its faces consist of 24 identical kites that have the same number of neighboring faces. From this, we can immediately deduce that there are exactly 24 ways to rotate one face onto another (without reflections). The rotation group of the solid is of order 24. The rotation order describes how many rotations exist that rotate the solid onto itself. This is a reformulation of the question we wanted to answer. So there 24 orientations for our 26 directions.

There exist only seven finite rotation groups in three dimensions, and they define all possible rotations for a finite set of points around the origin (for points that do not lie on a plane). The octahedral rotation group is the only rotation group that includes the main axes as orientations. The deltoidal icositetrahedron belongs to this rotation group. Thus, it is not possible to keep our constraint requiring orientations along the main axes and increase the number of orientations. If we give it up, we can use the vertices of an icosahedron as direction set and have a rotation order

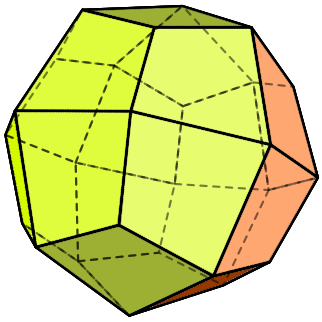


Figure 3.13: A deltoidal icositetrahedron

of 60. Since we do not want to do that, 24 different orientations is all that is possible. Note that the direction set can be enlarged by using subdivision, but this does not change the rotation order. In fact, all 24 orientations could be determined by just six directions: the corresponding solid is the cube. However, if we only used six directions, each of the probe samples would have to cover a sixth of the unit sphere to sample the whole environment, which is much too coarse. In the end, using 26 directions is a trade-off between simplicity and sufficient sampling, and we conclude that we cannot improve the number of supported orientations without using a different algorithm. For more information, see [CM80] and [Wik12].

3.4.2 Determining probe directions

It is useful to sample as little redundant information as possible to keep the number of probe samples small. We voxelize models to make sure we only generate probes at necessary positions. The subset of the direction set which we choose to generate probes for is the other important mean to reduce the number of probes without trading in precision. If we can sample a closed surface with the optimal amount of probes, we will in average only need about half of all possible 26 probe directions. If we do not, we will create twice as many probes. There are several possibilities to restrict the direction subset for each probe position:

Using probe positions

We can automatically generate directions for a probe position by using the probe position itself, which is relative to the model's center. For this, we compute the dot product between a possible direction and the probe position and dismiss it if it lies in the negative half-space of the normal. This works well for small objects but breaks for bigger objects that are not square.

For such objects, the relative position vector does not point outward any more but points along the object. This causes the issue.

Using averaged normals

A better method is to choose probe directions based on the average normals that the voxelizer computes. The advantage of using averaged normals is that we can use their length to measure the variance. If an averaged normal is unit length, we know that all triangles that were sampled

into that voxel had the same normal, so we can use all directions that lie in the same half-space as the normal. On the other hand, if an averaged normal is near zero, we know that the triangles that were sampled into the voxel had diverging normals , so we need to include more directions than just the positive half-space of the averaged normal.

For averaged normals that are not unit-length, we have found the following formula to work: a direction $\boldsymbol{d}$ is dismissed for averaged normal $\overline{\boldsymbol{n}}$ if

$$\boldsymbol{d} \cdot \overline{\boldsymbol{n}} < \|\overline{\boldsymbol{n}}\| \ (\|\overline{\boldsymbol{n}}\| - 1). \tag{3.14}$$

This works well in practice and generates too many directions at most. Since the voxelizer uses conservative rasterization even small triangles can spawn multiple voxels, which results in creating too many probes. This is worsened by the coarse voxelization resolution we use.

Using neighborhood information

Another approach we have tried is to determine good directions by using the neighbors of a voxel: if a voxel has a neighbor in a certain direction, we do not create a probe looking into this direction because it would only add redundant probe samples as that neighbor will create a probe for this direction itself. If there is no such neighbor, we create a probe looking into the direction.

This works well to avoid redundant probe direction in cases where voxelization creates unnecessary voxels. On the other hand, it also suffers from creating too many probes as it also generates probes for "back-facing" directions.

Using both averaged normals and neighborhood information

To solve this, we can merge both algorithms to reap the best of them:

1. the averaged normals are used to cull "back-facing" directions; then,
2. the neighbor voxels are used to remove redundant directions.

Specifically, we first check if the neighbor in the specific direction creates a probe for the direction. If so, we ignore the direction because the neighbor covers it. If not, we keep it.

Summary

Using averaged normals only works when the geometry is closed and the averaged normals are able to reflect this. Furthermore, it depends on the thresholding equation (3.14) to include sufficiently many directions.

3.5 Probe matching

The naive implementations shown in section 2.2 are too slow to work with datasets that are generated from Shadowgrounds Survivor levels.

The naive approach tries to match every probe sample of the query volume with every probe sample of the sampled model, resulting in $\Theta(n\,m)$ tests for n query probe samples and m model probe samples. This is effectively a **nested-loop join** of the two probe sample sets.

First optimization

As a first optimization of the naive approach, we employed a bucketing scheme on the probe samples in a preprocessing step. We create buckets for each occlusion value and distribute the probe samples into the different buckets. When we match two sets of probe samples, we create "job" lists with pairs of occlusion buckets that are within the tolerance value for the specific query. This job list is used to parallelize the matching process. To avoid race conditions, independent sets can be identified and each one can be executed concurrently. Alternatively, the score buffers can be duplicated for all threads and reduced in the end. We have implemented the latter because memory usage was not an issue and the time required for the reductions is negligible.

As another preprocessing step, we sort the probe samples in each bucket by distance. This allows us to run over the ranges using a skip pointer to reduce the number of comparisons from $\Theta(n\,m)$ to $\mathcal{O}(n\,m)$. At best, we only need n comparisons when we never have to jump back. This is an optimized **merge-join** algorithm.

Probe compression

We assume that there are regularities in the data: probe samples for the same probe will, to some degree, have similar colors, distance and occlusion values. This correlation can be exploited to compress the probe samples by merging groups with similar values (up to a small tolerance) into representative probe samples at the median of the range. This compression is not lossless, but by using a very small tolerance value the error becomes negligible.

Second optimization

Our second optimization is based on the following observations:

- the merge-sort approach is not fast enough for scenes with many models and bigger query volumes;
- the tolerance values that we use are not very small;
- the probe sample sets are rather sparse with concentrated clumps of probe samples, and the algorithm spends a lot of time skipping over probe samples that do not match;
- the bidirectional query only has to check whether a query probe sample can find a matching probe sample in the model probe samples and vice-versa, and not which one; and
- the configuration query only needs to access the probe's positions to calculate the target cell.

We decided to treat each of the two query types (bidirectional and configuration) individually and trade absolute correctness for speed: Instead of using the full range of color, distance and occlusion values, we quantize them to fit into a packed integer. We use $3\times4\times4$ bits or $4\times5\times5$ bits for the CIELAB colors. L has possible values between 0 and 100, a and b between -100 and 100. By using these numbers of bits, we can split the color ranges uniformly. For occlusion values, we use 2 or 3 bits, and for distances between 3 and 5 bits. All in all, a probe sample can be quantized into a **packed probe sample** that only needs 16 to 22 bits depending on the needed quality of the quantization. A regular probe sample needs 64 bits.

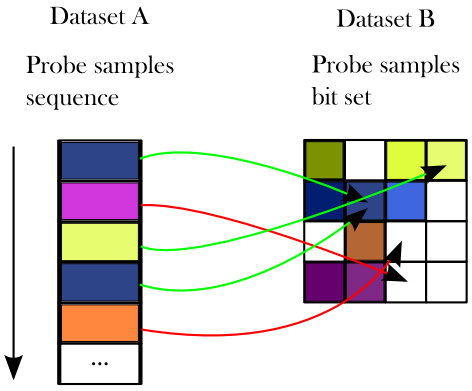


Figure 3.15: Bidirectional match query: matching dataset A with the dataset B. Only the matches of dataset A are counted. The same is done vice-versa as well to count the matches in dataset B (not shown). Unset elements in the bit set are drawn white. The colors represent the different packed probe samples. A green arrow means that the look-up was positive, a red one that it was negative. The match count for the shown probe samples is 3.

Bidirectional query

For configuration queries, we use these packed probe samples to speed up matching significantly: we splat all probe samples into a bit set that is directly addressed using the packed probe sample. A bit set for a single sampled model or query volume only needs between 8 KiB (2^{13} bytes) and 512 KiB (2^{19} bytes) depending on the quantization. This bit set only needs to be created once when the probe samples are sampled and can be stored in the probe database for all sampled models. The bit set for the query's probe samples can be created on the fly.

We also create a sequence of packed probe samples for the query volume and for all sampled models. This sequence can be stored in the probe database for all sampled models as well. To calculate the number of one-way matches in a bidirectional query, we run through these sequences once for the query and once for the sampled model, and look up every packed probe sample in the bit set of the sampled model, respectively the query. If it exists, we have found a matching probe sample, if not, there is none. We can count these matches and calculate the score. See figure 3.15 for an illustration. This takes exactly $2(n+m)$ operations and no comparisons at all. Moreover, to increase cache coherence of look-ups, the sequence that contains the packed probe samples can be sorted when it is constructed.

However, we lose some of the correctness of the previous approaches. Different tolerances for probe matching can be supported by using quantizations with other bit numbers. Nonetheless, this suffers from varying effective tolerances at the borders of the quantization buckets.

We have solved this partially by using a finer quantization and splatting a probe sample into multiple locations during the construction of the bit set. This effectively shifts the tolerance comparisons to the bit set. For the query probe samples, which have to be processed at run-time, this increases the run-time of every query, but the additional processing costs less operations than performing multiple look-ups. Using a finer quantization increases the size of the bit set and can

thus increase the chance for cache misses. This is not a huge problem on current processors for the sizes we use.

Importance-weighted bidirectional query

For this query, we need to use an additional look-up to retrieve the color's importance weight using the packed probe sample. By using the same quantization for the color counters, we can use a simple bit-mask and a bit-shift to compute the color bucket index. Otherwise, the algorithm stays the same.

Configuration query

Configuration queries cannot use the bit sets directly: for each matched probe sample, we need to know the position its probe to calculate the target cell.

During preprocessing, we create a flat immutable hash multi-map that maps packed probe samples to their probe positions. We create one such map for each probe direction. It contains all probe samples for that direction. Furthermore, we create a bit set for each direction as well to avoid unnecessary look-ups in the hash map, and we pre-transform all probe positions for all orientations to avoid matrix multiplications later on.

Since every direction is rotated exactly onto another direction by an orientation, we can simply match the probe samples for probes in one direction against the probe samples that point in the rotated direction to count all matches that support a certain orientation. We do this by iterating over all possible orientations and also over all possible directions. We determine the rotated direction and use the hash maps to speed up the look-ups. For each matched probe sample pair, we immediately know their probe positions because they are stored as values in the hash multi-map. This is considerably faster than the other implementations. See figure 3.16 on the facing page for an illustration.

To support different tolerances independent of finer quantization, we splat the same probe multiple times into the hash map. This is not optimal and could certainly be improved.

These algorithms are in effect optimized versions of the **hash-join** and **hash semi-join** algorithms.

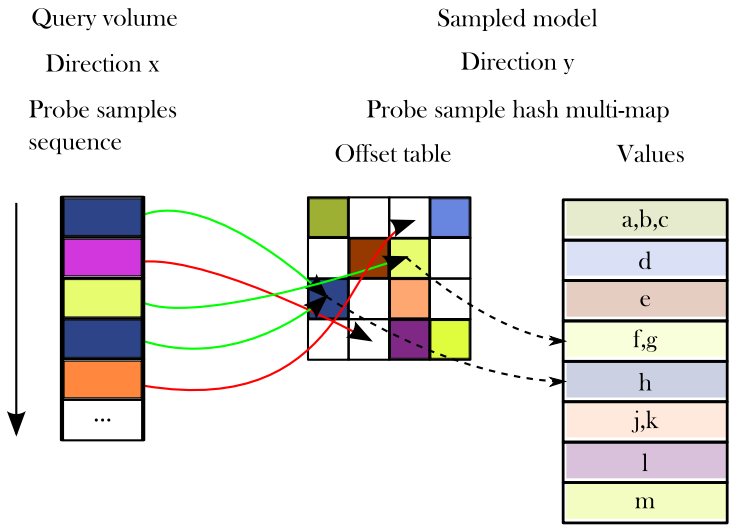


Figure 3.16: Configuration query: matching the query volume with the sampled model. The rotation that belongs to an orientation rotates direction x onto direction y (not shown). All probe samples for probes that look into direction x in the query volume are matched with the probe samples of the sampled model that look into direction y. Look-ups are optimized using a hash map, which stores the rotated probe positions for the orientation as values.

Chapter 4

Results

So far we have only described how everything works from a conceptual point of view. It is time that we present some results to ascertain how well Assisted Object Placement and its implementation work.

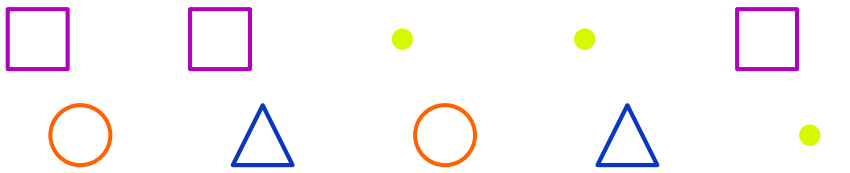
4.1 Validation using artificial scenes

We look at artificial scenes first because we can easier verify the results our algorithms return. We look at the neighborhood context first, and then at the probe context.

4.1.1 Neighborhood context

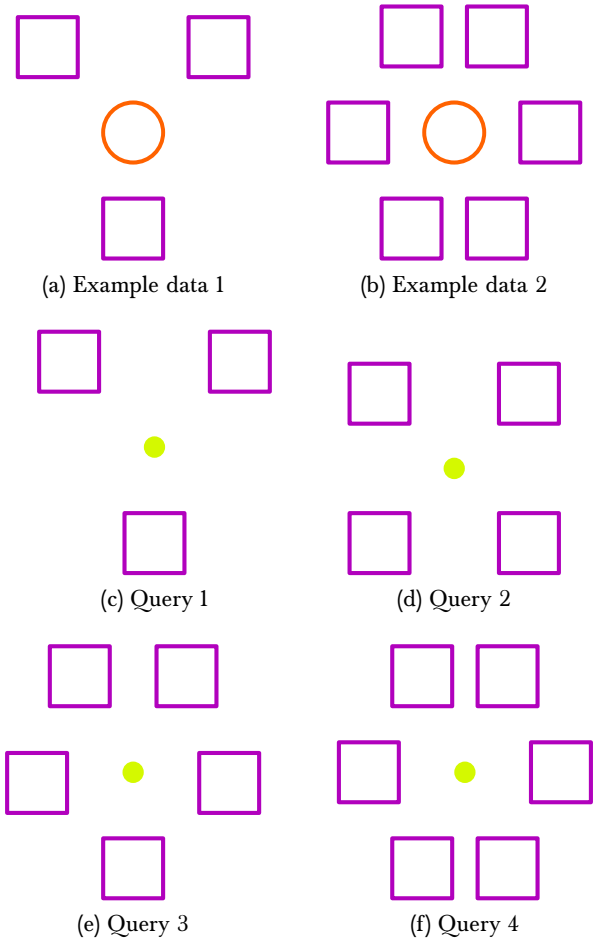
Scene 1

The first validation scene is shown in figure 4.1. We load the simple datasets in (a) and (b) into one neighborhood database and perform queries on the datasets (c), (d) and (e). All similarity measures correctly suggest □ for (c) and (d), and ○ and △ with equal probability for (e).



(a) Example data 1 (b) Example data 2 (c) Query 1 (d) Query 2 (e) Query 3

Figure 4.1: Validation scene 1. • marks the center of the query volume.



(a) Example data 1 (b) Example data 2

(c) Query 1 (d) Query 2

(e) Query 3 (f) Query 4

Figure 4.2: Validation scene 2. • marks the center of the query volume.

Scene 2

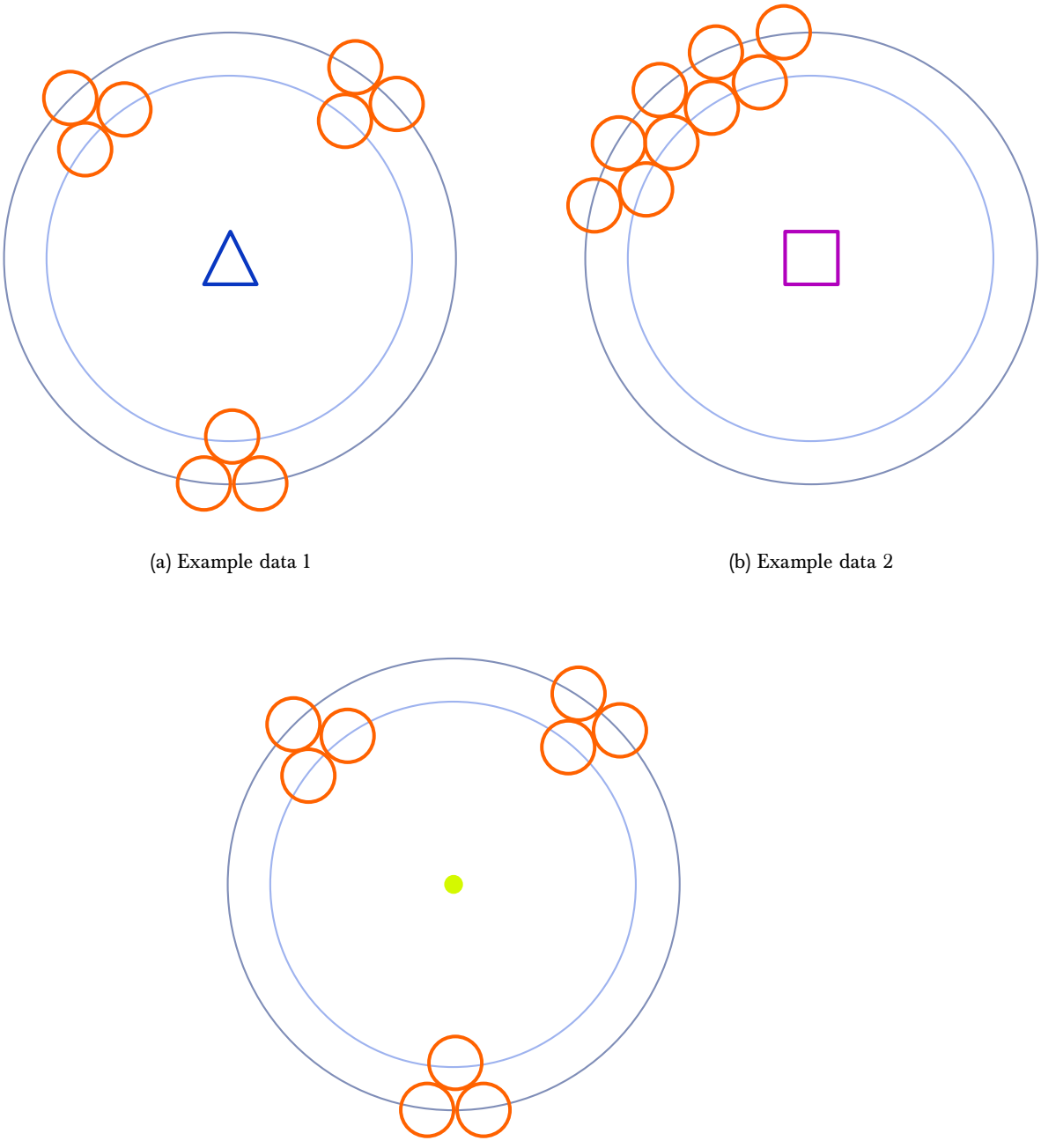
The second validation scene is shown in figure 4.2. We load the simple datasets in (a) and (b) into one neighborhood database and perform queries on the datasets (c), (d), (e) and (f). All similarity measures correctly suggest ○ for (c) and (f), and □ and ○ for (d) and (e) with lower probability.

Scene 3

The third validation scene is shown in figure 4.3 on page 62. We load the simple datasets in (a) and (b) into one neighborhood database and perform queries on dataset (c). It shows a failure of the matching algorithm: △ and □ will both be suggested with equal probability because the matching algorithm only takes distances into account and both example datasets have the same number of objects in each distance.

Jaccard index vs normal and importance-weighted Rand measure

The fourth validation scene uses the same datasets as the second one. They are shown in figure 4.2 on the facing page. We load the simple datasets in (a) and (b) into one neighborhood database and perform queries on an empty dataset (which is not shown). Only the query that uses the Jaccard index computes a probability of 0 for □, △ and ○. The other similarity measures compute similarities $\neq 0$. The Jaccard index returns the expected results.



(a) Example data 1

(b) Example data 2

(c) Query

Figure 4.3: Validation scene 3

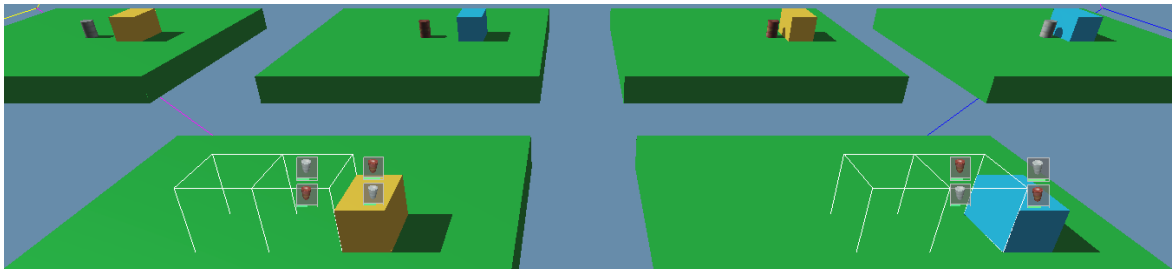


Figure 4.4: Validation scene 1

4.1.2 Probe context

Scene 1

The first validation scene shows that distance and color are important properties to distinguish between different surroundings. For an overview of the scene, see figure 4.4. From left to right at the top row, we have:

- a gray barrel that is 3 units away from an orange box;
- a red barrel that is 3 units away from a blue box;
- a red barrel that is 1 units away from an orange box; and
- a gray barrel that is 1 unit away from a blue box.

Below them, we see an orange box and a blue box. Each has two query volumes next to them. In succession from left to right, we see that the correct objects are chosen for each of the query volumes:

- the gray barrel is suggested for the query volume that is farther away from the orange box;
- the red barrel is suggested for the query volume that is next to the orange box;
- the red barrel is suggested for the query volume that is farther units away; and
- the gray barrel is suggested for the query volume that is next to the blue box.

When we do not sample distance or color, both objects are suggested with equal probability.

Scene 2

The second validation scene, which can be seen in figure 4.6 on page 65, shows that the configuration query works. Figure 4.6a shows the scene:

- gray barrels have been stacked next to the lit side of a lime green wall; and
- a group of red barrels has been placed next to the shadowed side of the wall.

Two query volumes are shown, each on a different side of the wall with a candidate list next to them that was generated by a bidirectional query.

Table 4.5 on the next page shows the exact results for both queries. Notice that the importance-weighted bidirectional query generally computes lower scores because each sampled instance is expecting its group neighbors but does not find them and consequently gets awarded no score for them. The probes that match are the ones that point towards the floor or the green wall. Both colors are more frequent in the scene and thus awarded a lower score which results in lower

	Left side	Right side
Gray barrel	0.59	0.37
Red barrel	0.23	0.75

(a) Results for both query volumes using a bidirectional query

	Left side	Right side
Gray barrel	0.46	0.31
Red barrel	0.15	0.69

(b) Results for both query volumes using an importance-weighted bidirectional query

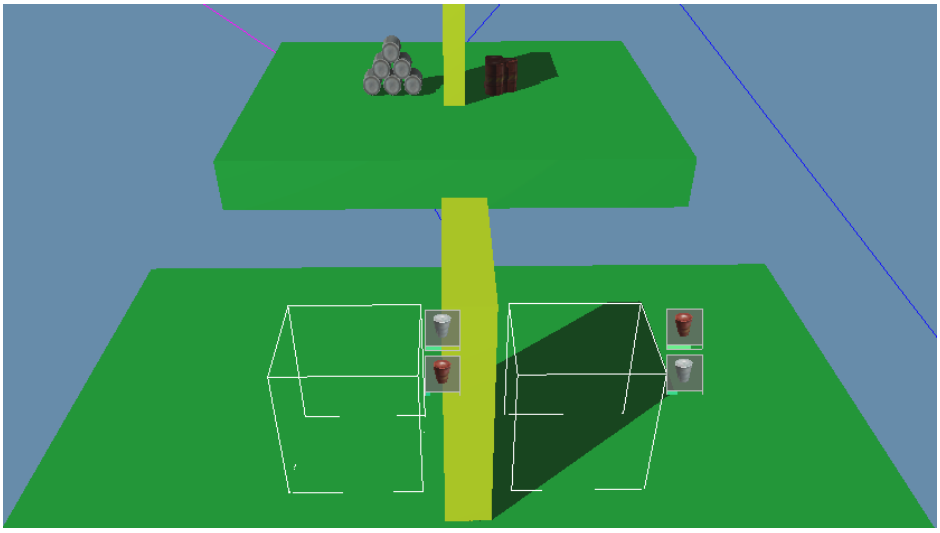
Table 4.5: Query results for the second validation scene

overall score compared to the bidirectional match query. If another barrel of the right type had been placed inside the query volume, the score of the importance-weighted bidirectional query would be bigger than the one of the normal query.

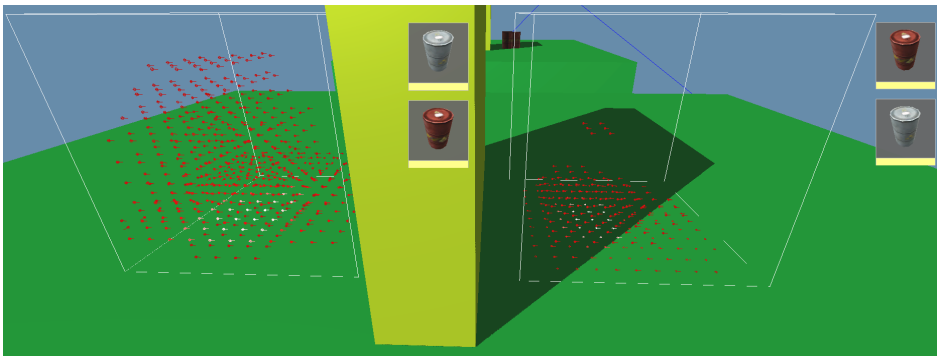
Figure 4.6b on the facing page shows the likelihood field for the best candidates in each query volume. The white arrows have the best score. In figure 4.6c the suggested best candidates have been placed using the suggested position and orientation of the configuration query.

Scene 3

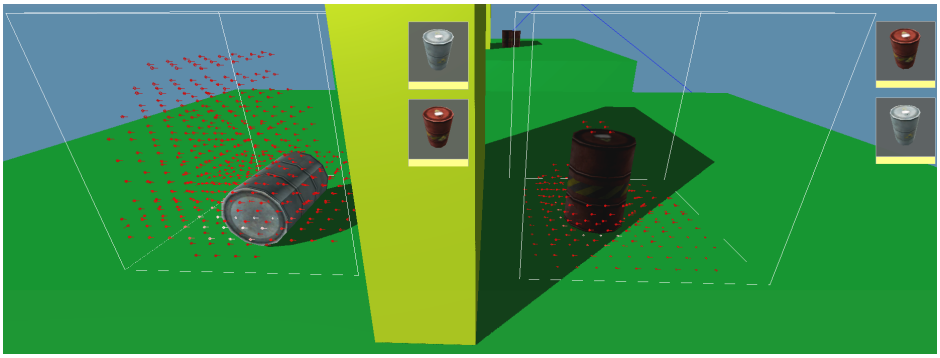
The third validation scene shows a case for which the configuration query fails. Figure 4.7 on page 66 shows a red barrel on a gray floor with a light green wall next to it. In front of them, we see a green floor with a gray wall on it and a query volume between them. A configuration query has been performed on it and we have a good match with a red barrel (the color tolerance has been increased for this example) that is placed on the gray wall because it matches the floor of the sampled instance. This is not what a level designer would expect.



(a) Overview and candidates. The candidates for the importance-weighted bidirectional query are displayed next to the query volumes.



(b) Likelihood fields for the best candidate in each query volume. The white arrows have the best score.



(c) Likelihood field for the best candidates together with the chosen suggestion.

Figure 4.6: Validation scene 2

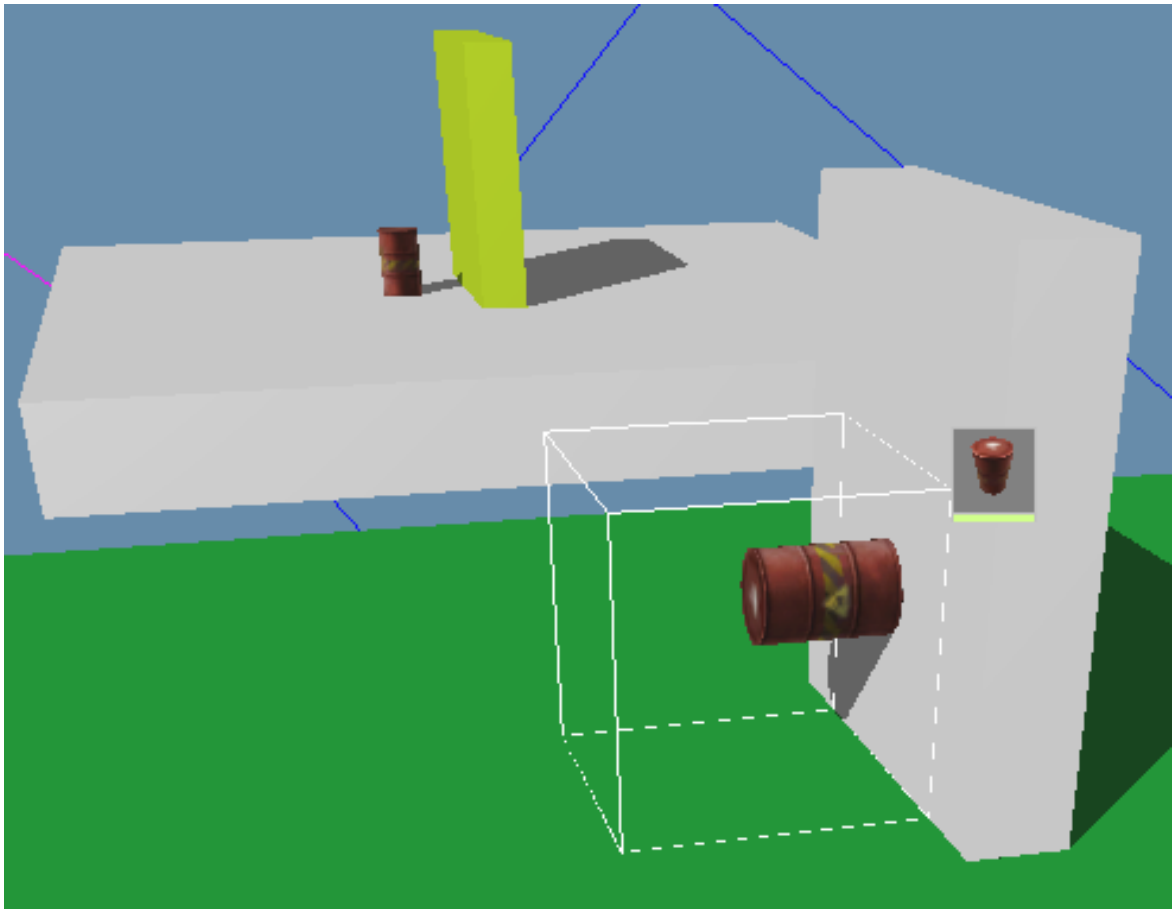


Figure 4.7: Validation scene 3

4.2 Validation using Shadowgrounds Survivor

It is important to verify that Assisted Object Placement also works in real-world use-cases to show it is applicable beyond artificial scenes.

Validating an algorithm that generates data is difficult. Assisted Object Placement suggests placements for new objects. How can we evaluate the quality of the suggestions if we have nothing to compare it with?

[YYW+12] uses surveys, but this is difficult to do in our case. In theirs, it works because they create small office or dining room scenes that everybody can evaluate. To evaluate the placement of objects in a game like Shadowgrounds Survivor using a survey, those who participate in it would need to know the game and even the level itself to be able to say if an object has been placed correctly or not. Games do not follow a real-world logic; they do not try to be as realistic as possible: they strive to be fun. For this, they bend the rules of logic and place objects at places that would not make sense normally. Thus, it would be hard to remove the bias introduced by the

	# models	# instances	avg rank (model frequency)
marine02_road	94	1396	17.15
marine01_wakeup	150	2066	21.10

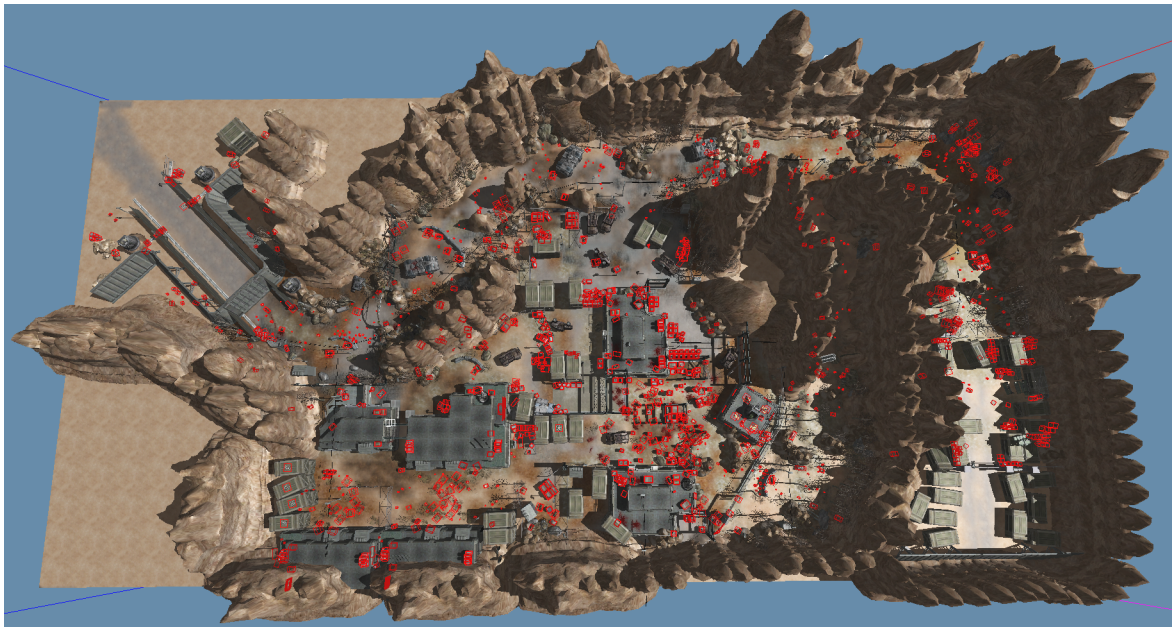
Table 4.8: Information about the sampled models in the Shadowgrounds Survivor levels being used for validation. The average rank for the model frequency is calculated by sorting the models by their instance count and calculating the average rank of the sorted list.

level itself. Even if the algorithm learns perfectly from the level, the placements could be marked as not fitting by a reviewer because the already existing level data does not match the reviewer's experience.

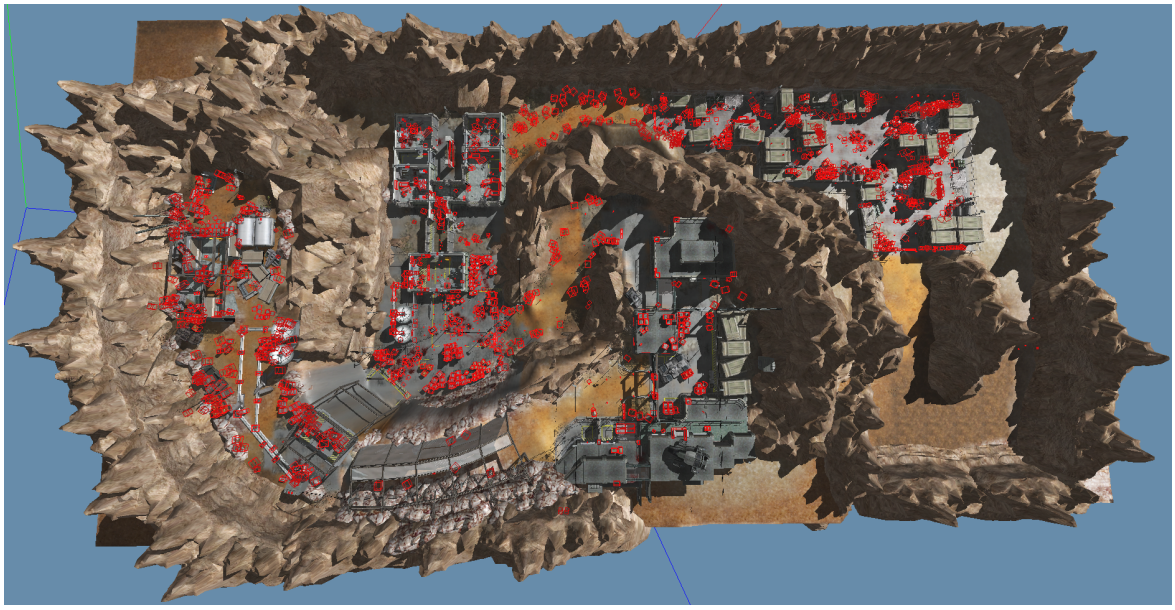
Our idea is to validate our algorithm with the level itself. We first learn from the whole level. Then, we remove one instance at a time and create a query at its old position and verify whether the model is suggested as candidate. We can use the rank of the model in the candidate list as score for that placement. The higher the rank, the worse the algorithm performed. We can compute the average of this rank for all instances in a level to get an estimate for how well it performs. To evaluate the quality of this approach, we can use the average rank of a dummy algorithm that suggests all models in the scene in order of their relative frequency as comparison.

It is obvious that the results should be very good for this approach since we more or less match data that already exists in the database with itself. Nonetheless, we only sample the scene with a very coarse grid resolution and quantize the data further, so this is a good initial test. Moreover, we can add a random shift to the positions at which we perform the query. This adds realism to the test, but is also a source for errors caused by the validation process itself. The shift ensures that we do not just test data from the database against itself, but on the other hand it also adds an additional error because the instance might very well not be the best model for the shifted position. This is especially true for areas with many props. Furthermore, we learn from the whole scene, and local areas of high object variance can throw off our algorithm even though the relative frequencies stay the same—and with them the average rank we compare our algorithm with. This could be fixed by choosing the query position using more complicated approaches, but then it is again not certain if they work better and how to validate such an hypothesis. For this reason, we simple add a small random shift to the query position for neighborhood queries and increase the query volume size for probe queries

We look at the neighborhood context first, then at the probe context, and finally at the combination of both. We validate Assisted Object Placement using the levels marine02_road and marine01_wakeup from Shadowgrounds Survivor. All objects with a diagonal length smaller or equal 2.7 units were selected and sampled into the neighborhood and probe database for validation. Figure 4.9 on the following page shows the two levels. All sampled instances have been highlighted. Table 4.8 gives some basic information about the sampled models in the two levels.



(a) `marine02_road`



(b) `marine01_wakeup`

Figure 4.9: Overview maps of the two Shadowgrounds Survivor levels that are used for validation. All sampled objects (objects with diagonal length ≤ 2.7) are highlighted.

Max random shift	2			4			8		
Max distance	10	20	60	10	20	60	10	20	60
Rand measure	0.38	0.43	0.46	1.16	1.20	1.23	3.38	2.90	2.91
IW measure	0.40	0.42	0.45	1.23	1.21	1.21	3.57	2.92	2.90
Jaccard index	0.38	0.43	0.46	1.14	1.20	1.23	2.86	2.88	2.90

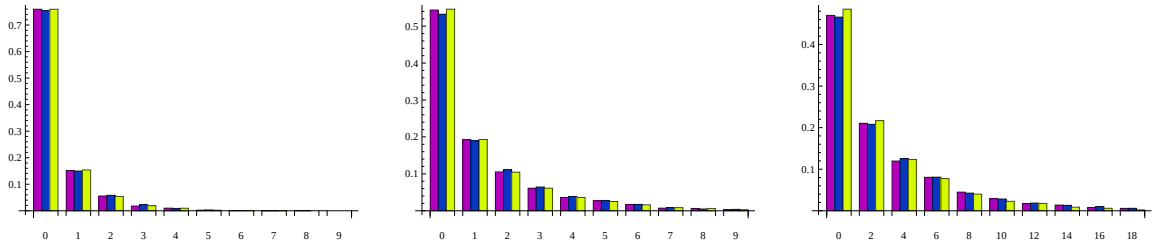
Table 4.11: Average ranks for the Shadowgrounds Survivor level `marine02_road` for different maximum distances and maximum random shift values. IW stands for importance-weighted.

4.2.1 Neighborhood context

We have validated the neighborhood context using the approach described above for different maximum distances and different random shift radii. The query tolerance was set to the random shift radius to allow the algorithm to make up for it. Table 4.11 shows the calculated average ranks, and figure 4.10 on the following page shows rank histograms for the different settings.

First, we notice that the rank results are very good even for high random shift radii. Compared to the average rank that we computed using the model frequencies, these results are even more stunning. Second, we can see that the Jaccard index outperforms both other measures when using a maximum distance of 10. With increasing maximum distance, the values do not differ much anymore. Third, the average ranks do not improve for higher maximum distances: they gets worse. We suspect the reason for this is the added noise, respectively the overfitting that occurs when the algorithm tries to match far-away neighbors that have no real correlation to the instance at hand.

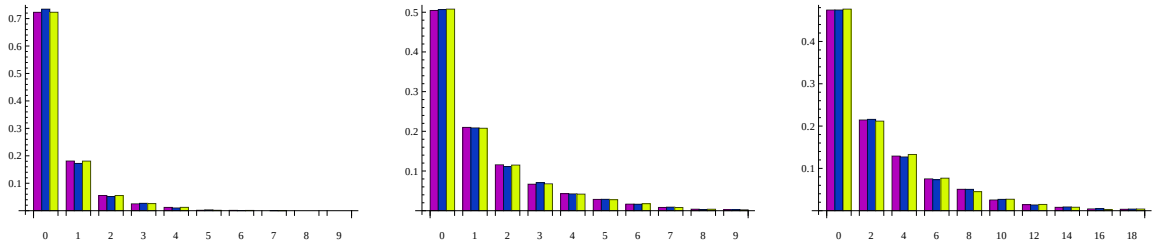
The histograms show that the quality of the measures is again very similar. They all falloff quickly though the "tail" grows longer for higher random shift radii, which was to be expected, and higher maximum distances, which again points towards the conclusion that the added "noise" created by the increased maximum distance negatively affects the algorithm. The results show that the right model is among the first few candidates in most cases.



(a) Max distance: 10, random shift radius: 2

(b) Max distance: 10, random shift radius: 4

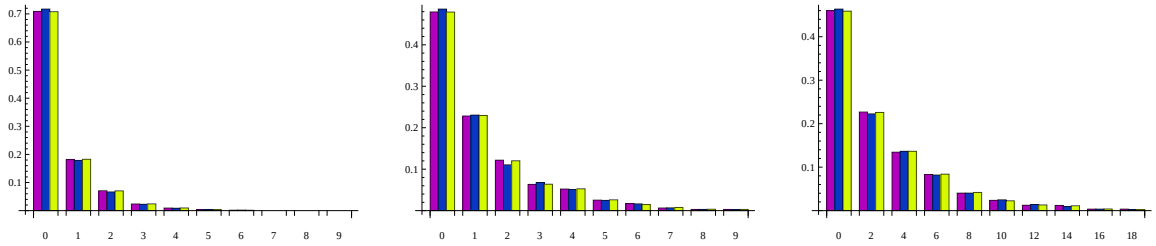
(c) Max distance: 10, random shift radius: 8



(d) Max distance: 20, random shift radius: 2

(e) Max distance: 20, random shift radius: 4

(f) Max distance: 20, random shift radius: 8



(g) Max distance: 60, random shift radius: 2

(h) Max distance: 60, random shift radius: 4

(i) Max distance: 60, random shift radius: 8

Figure 4.10: Neighborhood rank frequencies for the Shadowgrounds Survivor level `marine02_road` for different maximum distances and random shift radii. The Rand measure is shown purple; the importance-weighted measure is shown in blue; and the Jaccard index is shown in green.

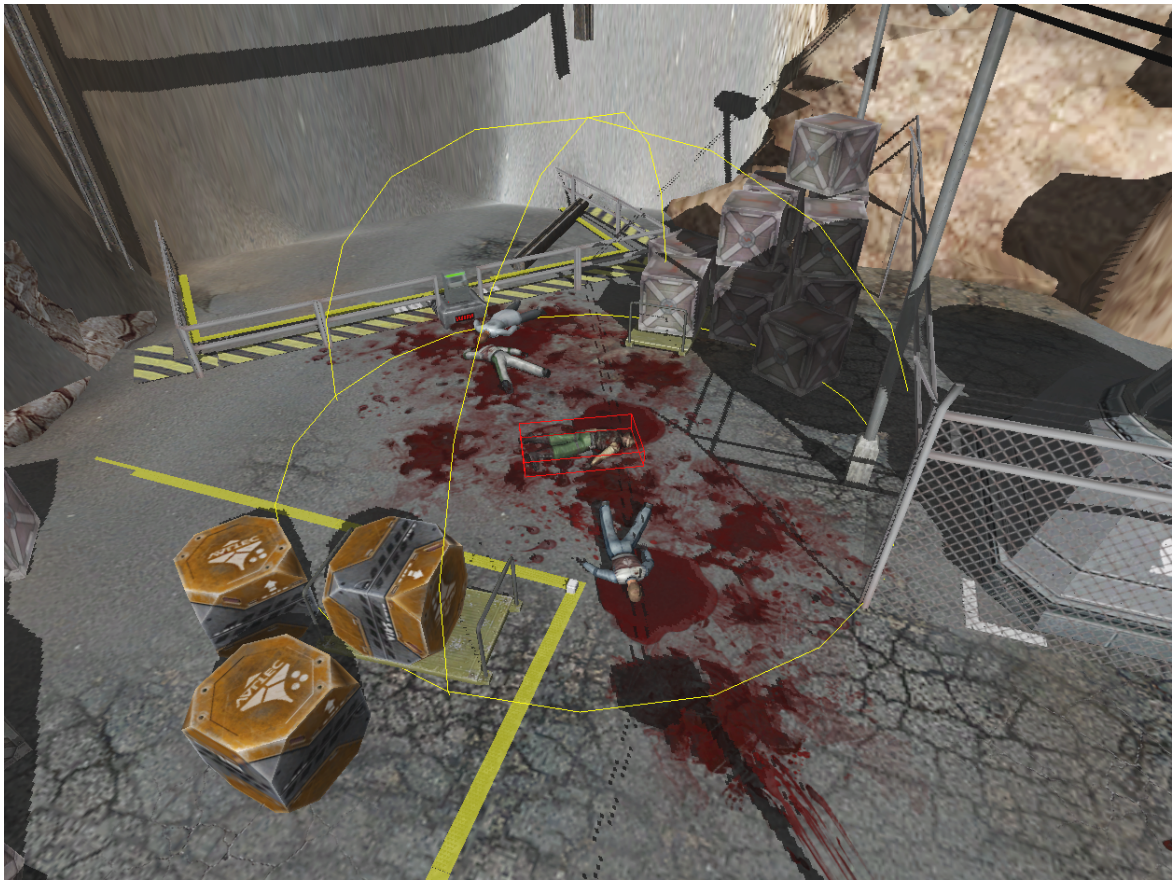


Figure 4.12: Visualization of the maximum probe distance (5 units). The selected corpse is the center of a sphere with radius 5.

4.2.2 Probe context

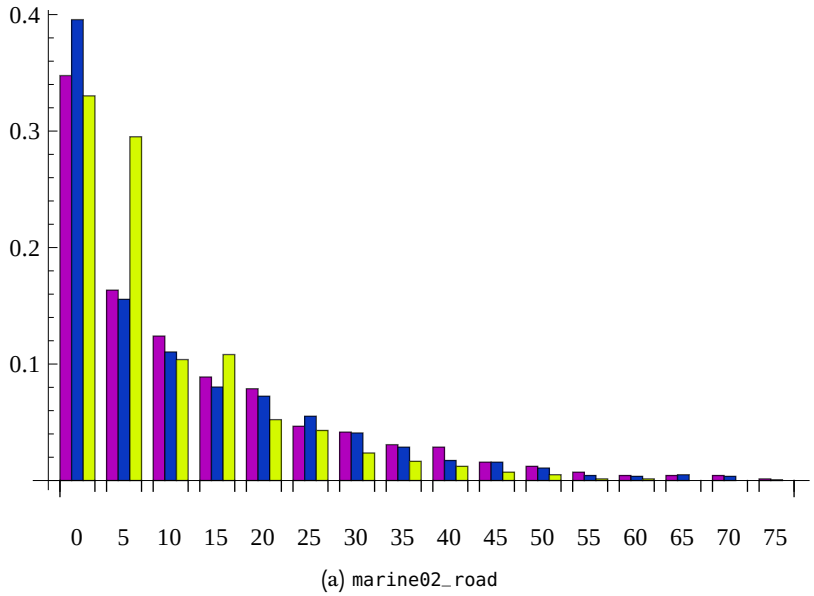
We have validated the probe context in Shadowgrounds Survivor using the bidirectional query, the configuration query and the importance-weighted bidirectional query in the levels `marine02_road` and `marine01_wakeup`. The maximum distance was set to 5 units. See figure 4.12 for a visualization of this maximum distance. The query volumes were chosen to contain an instance's origin and otherwise fit tightly to its bounding box. We have also validated `marine02_road` using query volumes of a fixed size of 5 to see how the algorithm copes with more variance.

Table 4.13 on the following page lists the reached average ranks, and figures 4.14 and 4.15 on page 73 and on page 74 show the rank histograms for our validation tests. The results are not as good as the ones for the neighborhood context. Nonetheless, we see that the importance-weighted bidirectional match query and the configuration query achieve a better average rank than the uniform bidirectional match query. The configuration query performs very well for tight volumes but performs worse for bigger volumes as can be seen in figure 4.15. Almost 10% less models have

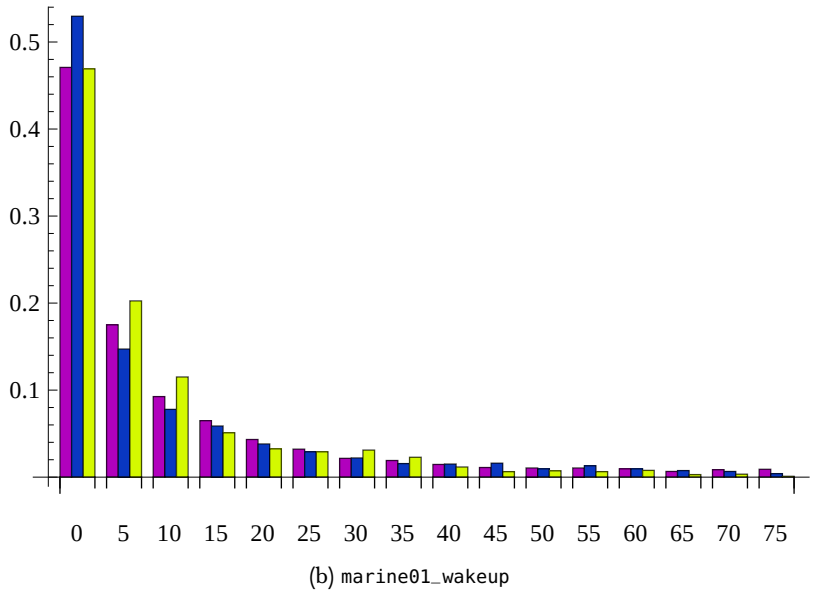
Level	marine02_road	marine02_road (5)	marine01_wakeup
Uniform bidirectional match	14.1	17.7	15.2
IW bidirectional match	12.9	17.6	14.3
Configuration	10.5	16.0	10.4

Table 4.13: Average ranks for different query types using no shift, a maximum distance of 5, and tight query volumes—except for marine02_road (5) which uses a fixed query volume of size 5. IW stands for importance-weighted.

rank of less than 5 when we use it compared to the other queries. When we use a query volume of size 5, the uniform and importance-weighted bidirectional match queries have worse average ranks than the model frequency. Only the configuration query has an average rank that is slightly better. All in all, this is alright as the probe context does not need to match as strictly as the neighborhood context. This has the benefit that overfitting will not become an issue and we will always get suggestions that make some sense. Most importantly though, we do not have to use the results from the probe context by themselves: we can combine them with the results from the neighborhood context.



(a) marine02_road



(b) marine01_wakeup

Figure 4.14: Probe rank frequencies for the Shadowgrounds Survivor levels marine02_road and marine01_wakeup for a maximum distance of 5, no shift, and tight query volumes. The bidirectional query results are shown in purple; the importance-weighted bidirectional query results are shown in blue; and the configuration query results are shown in green.

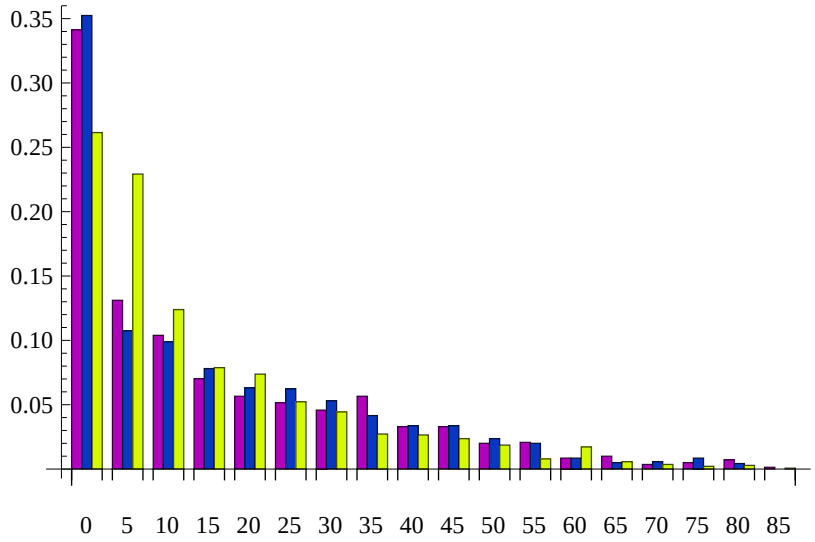


Figure 4.15: Probe rank frequencies for the Shadowgrounds Survivor levels `marine02_road` for a maximum distance of 5 and a query volume size of 5. The bidirectional query results are shown in purple; the importance-weighted bidirectional query results are shown in blue; and the configuration query results are shown in green.

4.2.3 Combined matching

For the combined matching, we have merged the results from the probe context with the results from the neighborhood context by multiplying them. We have three queries for the probe context, three different similarity measures for the neighborhood context, and three test scenarios: `marine02_road`, `marine01_wakeup`, and `marine02_road` with a query volume size of 5. The other two scenarios use tightly fitting query volumes again. The results of the neighborhood context have been computed without random shifts but with a fixed query tolerance of 5 to avoid perfect matches and to better resemble a real scenario. Table 4.16 on the next page shows the average ranks for the combined results. The results are amazing. The Jaccard index is one order of magnitude better than the other two measures in our tests. When we combine results, the configuration query and the importance-weighted similarity measure perform worst in all combinations, and the uniform bidirectional query and the Jaccard index best. This suggests that further research is needed to determine whether the results for the combination of the uniform bidirectional query and the Jaccard index hold in general. We do not know yet how to create further experiments to test this hypothesis.

	marine02_road			marine02_road (5)			marine01_wakeup		
	Rand	IW	Jaccard	Rand	IW	Jaccard	Rand	IW	Jaccard
Uniform bidirectional	7.14	11.0	**0.230**	7.34	12.6	**0.176**	8.85	12.3	**0.651**
IW bidirectional	8.09	11.0	**0.399**	9.81	14.4	**0.320**	9.79	12.5	**0.926**
Configuration	11.1	11.5	**4.40**	15.9	16.5	**5.83**	21.8	22.2	**15.2**

Table 4.16: Average ranks obtained from the combined results for all possible query and similarity measures. This is computed using the same datasets as table 4.13. The neighborhood results are computed without random shifts and a query tolerance of 5. IW stands for importance-weighted.

	Voxelization	Sampling
marine02_road	5.5	320
marine01_wakeup	8.4	520

Table 4.17: Timings for the voxelization all models with a diagonal length ≤ 2.7 and the probe sampling of all their instances. All times are in seconds.

4.3 Performance

We measured the time it took to voxelize all models with a diagonal length ≤ 2.7, to sample all their instances (using probes) in a level, which includes post-procesing the data and storing in the database, and to run the validation tests on them. For this, we have used a workstation with an Intel Xeon X5650 processor, 12 GB of memory and an NVIDIA GTX 560 Ti graphics card. The results are summarized in tables 4.17 to 4.19 on this page and on the next page. We can see that:

- voxelization and probe sampling scale linearly with the number of models and instances (since we use a size threshold for the models);
- neighborhood matching scales linearly in the number of instances; and
- probe matching scales linearly in the number of instances, and scales with the third power of the query volume size.

The timings in the tables are totals for all instances, so we need to divide by the number of instances first to see these relations. Last, for comparison, we have measured the time it takes to run a single query using the first optimized version of the bidirectional match query in marine02_road with a query volume size of 5: 280 seconds. The second optimization has resulted in a significant speed-up: it needs 14 seconds in total to perform all 1396 queries.

	Rand measure	Importance-weighted measure	Jaccard index
marine02_road	9.5	11	9.5
marine01_wakeup	38	44	39

Table 4.18: Timings for running neighborhood queries on all instances using different similarity measures with a maximum distance of 5 and no random shift. All times are in seconds.

	Bidirectional query	IW bidirectional query	Configuration query
marine02_road	1.2	5.4	900
marine02_road (5)	14	93	18000
marine01_wakeup	2.4	11	1500

Table 4.19: Timings for running bidirectional queries, an importance-weighted bidirectional queries and a configuration queries on all instances with the settings described in section 4.2.2. All times are in seconds.

Chapter 5

Conclusion

We have developed a comprehensive approach to assist level designers with placing objects automatically in a scene. It can work with existing level data without any augmentations, and it does not need any additional user input after calibrating the tolerances once for a game. Neither do we require the user to specify any additional constraints or rules for our algorithm to work. It supports learning from new placements by design, and it can even compute the best positions and orientations for the objects it suggests if desired. We offer several algorithms with different performance characteristics that can be used depending on the level of information needed. We have validated different procedures and determined the best ones for various scenarios and tested them with data from a real game. Additionally, we have examined concepts from information theory and attempted to motivate our course of action with a probabilistic model. Even though the performance of our more advanced algorithms does not allow real-time editing yet, the simpler approaches could well be implemented in current games and provide significant time saving when placing small objects. They can also assist in making levels more coherent when multiple level designers work on them by offering context-sensitive suggestions.

Our algorithms certainly exhibits some weaknesses, which we would like to improve: for one, probe matching by itself is still performing below its possibilities in our opinion and the queries could be tweaked to use even better scoring algorithms. In addition, the tolerances could be calibrated automatically and for each sampled model using an approach similar to [CLS10]. We could also improve performance by using more advanced data structures. For instance, perfect spatial hashing [LH06] could be used to optimize the hash tables we generate for configuration queries, or we could improve the algorithm that matches neighborhoods.

The probabilistic model could also be improved to create a fully inclusive model that can be used to derive algorithms that can place large objects automatically as well as generate totally new scenes.

Bibliography

[AMHH08] Tomas Akenine-Möller, Eric Haines, and Naty Hoffman. *Real-time rendering*. Peters, Wellesley and Mass, 3 edition, 2008.

[Bis06] Christopher M. Bishop. *Pattern recognition and machine learning*. Information science and statistics. Springer, New York, 2006.

[CLS10] Matthäus G. Chajdas, Sylvain Lefebvre, and Marc Stamminger. Assisted Texture Assignment. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*. ACM Press, 2010. Available from: http://www-sop.inria.fr/reves/Basilic/2010/CLS10.

[CM80] H. S. M. Coxeter and W. O. J. Moser. *Generators and relations for discrete groups*. Springer-Verlag, Berlin and and New York, 4 edition, 1980.

[Coz12] Patrick Cozzi, editor. *OpenGL Insights: [OpenGL, OpenGL ES, and WebGL community experiences]*. CRC Press, Boca Raton, 2012.

[CWW11] Matthäus G. Chajdas, Andreas Weis, and Rüdiger Westermann. Assisted environment probe placement. 2011.

[FH10] Matthew Fisher and Pat Hanrahan. Context-based search for 3D models. *ACM Trans. Graph.*, 29(6):182:1–182:10, 2010. Available from: http://doi.acm.org/10.1145/1882261.1866204, doi:10.1145/1882261.1866204.

[FRS+12] Matthew Fisher, Daniel Ritchie, Manolis Savva, Thomas Funkhouser, and Pat Hanrahan. Example-based Synthesis of 3D Object Arrangements. In *ACM SIGGRAPH Asia 2012 papers*, SIGGRAPH Asia '12, 2012.

[FSH11] Matthew Fisher, Manolis Savva, and Pat Hanrahan. Characterizing structural relationships in scenes using graph kernels. In *ACM SIGGRAPH 2011 papers*, SIGGRAPH '11, pages 34:1–34:12, New York and NY and USA, 2011. ACM. Available from: http://doi.acm.org/10.1145/1964921.1964929, doi:10.1145/1964921.1964929.

[Gam95] Erich Gamma. *Design patterns: Elements of reusable object-oriented software*. Addison-Wesley, Reading and Mass, 1995.

[Khe02] Leow Wee Kheng. Color Spaces and Color-Difference Equations, 2002.

[LH06] Sylvain Lefebvre and Hugues Hoppe. Perfect spatial hashing. *ACM Trans. Graph.*, 25(3):579–588, 2006. Available from: http://doi.acm.org/10.1145/1141911.1141926, doi:10.1145/1141911.1141926.

[MvEO94] M. Mahy, L. van Eycken, and A. Oosterlinck. Evaluation of uniform color spaces developed after the adoption of CIELAB and CIELUV. *Color research and application*, 19(2):105–121, 1994.

[NC08] H. Nemmour and Y. Chibani. New Jaccard-Distance Based Support Vector Machine Kernel for Handwritten Digit Recognition. In *Information and Communication Technologies: From Theory to Applications, 2008. ICTTA 2008. 3rd International Conference on*, pages 1–4, 2008. doi:10.1109/ICTTA.2008.4530078.

[NK03] Marcin Novotni and Reinhard Klein. 3D zernike descriptors for content based shape retrieval. In *Proceedings of the eighth ACM symposium on Solid modeling and applications*, SM '03, pages 216–225, New York and NY and USA, 2003. ACM. Available from: http://doi.acm.org/10.1145/781606.781639, doi:10.1145/781606.781639.

[NSMO10] Noor Aznimah Abdul Aziz, Siti Salwa Salleh, Daud Mohamad, and Megawati Omar. Investigating Jaccard Distance similarity measurement constriction on handwritten pen-based input digit. In *Science and Social Research (CSSR), 2010 International Conference on*, pages 1181–1185, 2010. doi:10.1109/CSSR.2010.5773712.

[PF05] Matt Pharr and Randima Fernando, editors. *GPU Gems 2: Programming techniques for high-performance graphics and general-purpose computation*, volume 2 of *GPU gems*. Addison-Wesley, Upper Saddle River NJ, 2005.

[PH04] Matt Pharr and Greg Humphreys. *Physically based rendering: From theory to implementation*. The Morgan Kaufmann series in interactive 3D technology. Elsevier, Amsterdam [u.a.], 2004.

[PM01] Yoav I. H. Parish and Pascal Müller. Procedural modeling of cities. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '01, pages 301–308, New York and NY and USA, 2001. ACM. Available from: http://doi.acm.org/10.1145/383259.383292, doi:10.1145/383259.383292.

[Ran71] William M. Rand. Objective Criteria for the Evaluation of Clustering Methods. *Journal of the American Statistical Association*, 66(336):p 846–850, 1971. Available from: http://www.jstor.org/stable/2284239.

[Ros07] Sheldon M. Ross. *Introduction to probability models*. Academic Press, Amsterdam and and Boston, 9 edition, 2007.

[RV96] R. Real and J.M Vargas. The probabilistic basis of Jaccard's index of similarity. *Systematic Biology*, 45(3):380–385, 1996.

[SAMO11] S.S Salleh, N.A.A Aziz, D. Mohamad, and M. Omar. Combining Mahalanobis and Jaccard to Improve Shape Similarity Measurement in Sketch Recognition. In *Computer Modelling and Simulation (UKSim), 2011 UkSim 13th International Conference on*, pages 319–324, 2011. doi:10.1109/UKSIM.2011.67.

[SS] Angelika Steger and Thomas Schickinger, editors. *Diskrete Strukturen*. Springer-Lehrbuch. Springer, Berlin [u.a.].

[Sto03] Maureen C. Stone. *A field guide to digital color*. Peters, Natick and Mass, 2003.

[TLL+11] Jerry O. Talton, Yu Lou, Steve Lesser, Jared Duke, Radom\'ır Měch, and Vladlen Koltun. Metropolis procedural modeling. *ACM Trans. Graph.*, 30(2):11:1–11:14, 2011. Available from: http://doi.acm.org/10.1145/1944846.1944851, doi:10.1145/1944846.1944851.

[VKW+12] Carlos A. Vanegas, Tom Kelly, Basil Weber, Jan Halatsch, Daniel G. Aliaga, and Pascal Müller. Procedural Generation of Parcels in Urban Modeling. *Comp. Graph. Forum*, 31(2pt3):681–690, 2012. Available from: http://dx.doi.org/10.1111/j.1467-8659.2012.03047.x, doi:10.1111/j.1467-8659.2012.03047.x.

[Wik12] Wikipedia. Point groups in three dimensions — Wikipedia, The Free Encyclopedia, 2012. Available from: http://en.wikipedia.org/w/index.php?title=Point_groups_in_three_dimensions&oldid=520667706.

[YYW+12] Yi-Ting Yeh, Lingfeng Yang, Matthew Watson, Noah D. Goodman, and Pat Hanrahan. Synthesizing open worlds with constraints using locally annealed reversible jump MCMC. *ACM Trans. Graph.*, 31(4):56:1–56:11, 2012. Available from: http://doi.acm.org/10.1145/2185520.2185552, doi:10.1145/2185520.2185552.