Master Thesis

Realistic Real-Time Landscape Rendering Using GIS Data

Quentin Kuenlin

Professor: Wenzel Alban Jakob

Supervisor: Jesse van den Kieboom

EPFL
Lausanne, Switzerland

August 17, 2018

Abstract

Esri's ArcGIS scene viewer is a web-based 3D real-time rendering of the Earth. Currently, satellite images and heightmaps are mapped on the globe which give good results at far and mid-range view distances. However as soon as the viewer get to close to the ground, the resolution of the satellite images is not good enough and the rendered view become pixelated.

Therefore, this Master Thesis investigates the possibility to enhance the rendering quality of natural landscapes at mid to close range. This was done in two parts. In the first one, machine learning algorithms were used to automatically detect the type of land cover based on satellite images and height information. Support Vector Machines and Decision Forest were tested with several configurations. The SVM with Gaussian kernel had a 82.36% of accuracy while the Decision Forest with 80 trees achieved a 98.18% of accuracy. The contribution of each features were tested and some of them were slightly reducing the global accuracy. In the end, the final classifier was a Decision Forest with an average accuracy of 98.76%.

In the second part, the classified data were used to guide the shader to render the correct materials. The possibilities to use physically based shading as well as global illumination in real-time rendering were demonstrated using the Disney Principled BSDF. A multi-scale material technique was proposed using several layers of textures. Transition between materials was investigated using a simple interpolation and a more complex height based blend. Finally, a full procedural simulation and rendering technique for the water was demonstrated.

Acknowledgement

I would like to thanks everyone at Esri R&D Center Zürich, especially Jesse and Johannes whose inputs on the project were greatly appreciated. A special thanks to everyone who took the time to test my project and gave me valuable feedback.

I would also like to thank my family, specifically my parents, who supported me during all my studies.

Contents

Chapter 1

Introduction

Being based on satellite imagery, the ArcGIS scene viewer depends on the quality of the images to be able to render realistic landscapes. While modern satellites can take pictures to a centimeter precision, mapping the entire earth with this kind of precision would take a long time, and also take a huge amount of memory. Currently, ArcGIS as a precision of up to 1.2 meters per pixels at zoom level 17 for most of the planet. This level of precision is good enough for far to mid view distances but as soon as we go to the ground level, the result is blurry, as we can see in Figure 1.1. Moreover, as the ArcGIS scene viewer only map the satellite images and height data on the terrain, there is a lack of realistic dynamic shading.

Figure 1.1: Views in ArcGIS scene viewe from far (left), mid (center), and close (right) distances.

Therefore the main goal of this Master Thesis is to investigate a way to increase the realism of the landscape rendering, without requiring a better satellite image quality. This will required the ability to classify the satellite images to know what kind of surface we need to render.

A multitude of study has been done in the field of land cover classification. However, the majority of them uses multi-spectral satellite images. These data can be freely access, at a low resolution, through the Sentinel-2 or Landsat satellites from the European Space Agency (ESA) and National Aeronautics and Space Administration (NASA) respectively. The main advantage of these multi-spectral data is that they give access to the infra-red and ultra-violet spectrum which are not available with classic visible spectrum images. However, in our case, we want to be able to classify our terrain using classic satellite images only. As the multi-spectral images are freely available, there exits only a few researches that do not use them. These research

demonstrate that it is possible to achieve a good level of accuracy with a low number of clearly distinguishable land cover.

Knowing which kind of surface we need to render will help increase the realism of the visualization, but is not enough to have realistic results. The ability to render realistic surfaces comes down to simulating the behaviour of materials and light. The interaction between light and surfaces is one of the most researched topics in computer graphic. Whether it is for offline or real-time rendering, there exists multiple techniques to shade objects. The early methods, such as the Phong shading, were based on observation of the real world and required a lot of work to make them look realistic. On the other hand, modern models are based on physical rules and can give realistic results with a lot less effort from the user, but are more computationally demanding.

With modern hardware, it is possible to render in real-time large landscape using physically based rendering. Indeed, as we can see in Figure 1.2, it is possible to obtain realistic real-time results using state-of-the-art rendering techniques. However, these landscape rendering methods rely heavily on high definition assets, such as rocks, trees, or bushes. Therefore they usually require a recent and powerful GPU to be able to run in real-time.

Figure 1.2: Screenshot of Tom Clancy's Ghost Recon® Wildlands (2017)[1].

In this Master Thesis, we will only focus on enhancing the existing surface by creating realistic materials for each type of land cover. As the terrain is composed of one unique mesh, all different material types, including water, will have to be handle in one shader.

.

[1]From `http://ghost-recon.ubisoft.com/wildlands`

Part I

Land Covers Classification

Chapter 2

Introduction

In this first part, we will see how we can use satellite images to segment the terrain in different land covers. Our goal is to be able to classify numerous different land cover types, such as snow, grass, water, rock, etc. The data provided by Esri that we can use for this purpose are satellite images of 256 by 256 pixels at a maximum precision of 1.2m per pixels as well as height data with a precision of 9.5m per pixels. The training of the algorithm and its inferring need to be as fast as possible while still having the highest accuracy possible.

For that purpose, we will first look at the most used algorithms for land classification and see how they performed in other studies. We will then explore the different features, beside the color component, that we can extract from the satellite images. Finally, we will test our classification tool using several configurations to find the most suitable classifier for our needs.

Because we are aiming at rendering only natural landscape, we will focus the classification on natural ground cover only. Moreover, we will focus only on mountain type landscapes such as the one we have in the Swiss Alps. Therefore, we chose seven ground cover commonly found in this type of landscape: rock, gravel, soil, grass, forest, snow, and water.

Chapter 3

Classification Algorithms

Several different classification algorithms can be used for land cover classification. In this chapter, we will briefly present the two most used algorithms.

3.1 Classification Methods

In the field of classification, there exist two main types of algorithms: supervised and unsupervised. In addition to these two methods, when classifying land covers based on images, we can either process the image pixel per pixel, or per group of neighbor pixels, called objects. Those 4 variants are presented below.

3.1.1 Supervised and Unsupervised Classification

Supervised classification needs to be trained before being able to classify images. This means that the underlying algorithm needs to be fed with pixels of a known class by the user, which is called the training phase. The classifier will then be able to infer the class of other pixels based on the known pixels.

Contrary to the supervised classification, unsupervised algorithms do not need any prior knowledge to be able to classify an image. This type of classification creates large groups of pixels, or objects, with similar features together and then creates a defined number of classes with these groups.

3.1.2 Pixel-Based and Object-Based

When using pixel-based classification, the classifier will classify each pixel individually. This does not mean that the classifier will not use the information of the surrounding pixels, however the classification of one pixel will not influence the classification of its neighbors.

Object-based classifier will first group neighbor pixels with similar features together. Each group will then be classified as one class for all underlying pixels. This method is particularly useful when classifying high resolution images. Indeed, with higher resolution, single object such as trees, cars, etc. can be formed by hundreds of pixels. It is thus better to use object-based classification in those cases.

3.2 Support Vector Machine

3.2.1 Overview

Support Vector Machines (SVM), first introduced by Cortes and Vapnik [1995], are non-probabilistic binary linear classifiers. When training, the algorithm will calculate a hyperplane that separates two classes. In most cases, there exists an infinite number of possible hyperplanes. However, most of them will be too noise sensitive as they are too close to the points. Therefore, the optimal hyper-plane is defined as the one that maximize the distance between itself and every point in the training set. Once the SVM is trained, new points can be classified by looking at which side of the hyper-plane they are.

3.2.2 Multi-Classes

As SVM are binary classifiers, they cannot, by definition, handle more than two different classes. However, this can be solved by combining several SVMs. Indeed, a one-against-one or a one-against-all strategy can be used to solve a multi-class problem. When using one-against-one strategy, a SVM will be created for each pair of classes, resulting in $C * (C-1)/2$ SVMs for C classes. In the other hand, when using one-against-all strategy, a SVM will be created for each class, resulting in only C SVMs. To classify a point using multiple SVMs, each one vote for the best fitting class and the most voted class is chosen.

3.2.3 Non-Linear Problems

As stated previously, SVMs can only separate two classes if a hyper-plane exists between all points in each of the classes. Such a hyper-plane might not exist for non-linear data such as depicted in Figure 3.1 (a). To be able to classify non-linear problems, we can transform the points using a function called "kernel trick". This function performs a defined operation on the points to make them linearly separable.

The most used kernel tricks are:

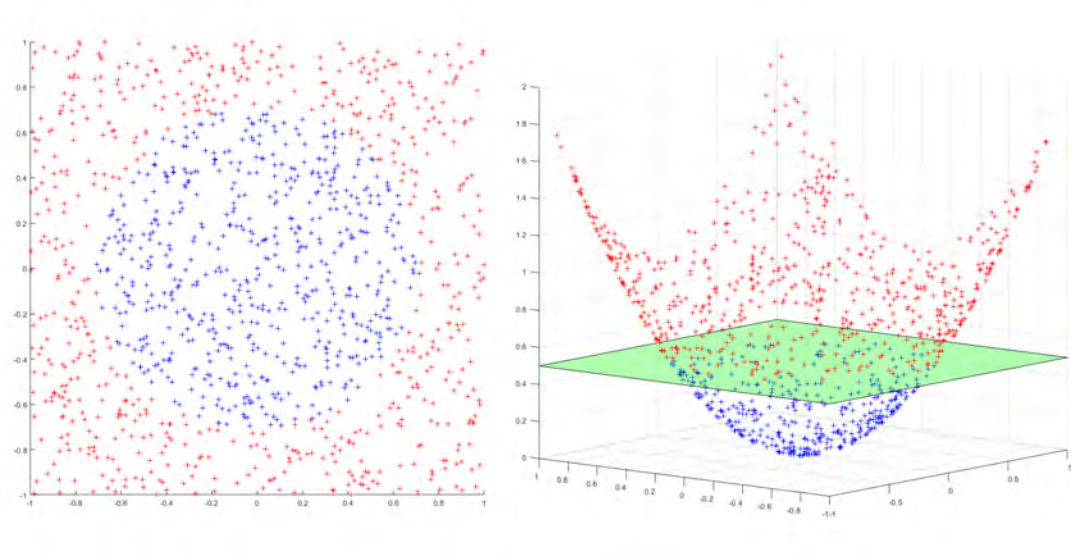
- **Linear:** $K(\vec{a}, \vec{b}) = \vec{a} * \vec{b}$
- **Gaussian:** $K(\vec{a}, \vec{b}) = exp\left(-\gamma||\vec{a} - \vec{b}||^2\right)$
- **Polynomial:** $K(\vec{a}, \vec{b}) = \left(\vec{a} * \vec{b}\right)^d$

3.3 Decision Forest

3.3.1 Overview

Decision Forests, also known as Random Forests, were first introduced by Ho [1995]. They are a group of multiple Decision Trees that were each trained differently. Decision Trees are tree-like structures that can be traversed to classify a point. Each branching in the tree represents a comparison to a feature, while each leaf corresponds to a possible class. Figure 3.2 shows an example of a Decision Tree.

To classify a point using Decision Forest, each Decision Tree in the forest votes for the best class. When every tree has voted, the class with the most votes will be the classification result. The advantage to use multiple Decision Tree instead of just one, is that it reduces their habit



(a) (b)

Figure 3.1: Example on how a Kernel trick can help to linearly separate two classes. (a) Samples in $\mathbb{R}^2$ that are not linearly separable. (b) Same samples transformed using a function $\mathbb{R}^2 \rightarrow \mathbb{R}^3$ which makes the classes linearly separable.

of over-fitting to the training set. Indeed, Decision Tree will be closely tied to the training set used. It is why having multiple trees trained differently will reduce over-fitting.

3.3.2 Construction

The construction of a Decision Tree is a recursive process. At each iteration, the algorithm finds the best feature that is able to partition the training set into two subsets of similar sizes and creates a branching with the found condition. Each subset is then processed independently of the other. When a subset contains only members of a single class, a leaf is created with that class.

To create a Decision Forest, a defined number of Decision Trees are created. Several approaches exist to grow different trees in a forest. The first one is to vary the weight of the training sample for each tree, making each sample more or less important depending on their attributed weight. A second approach is to change the tree growing algorithm and add some randomness in the process, for example randomly selecting the attribute to partition the set. A third option is to train each tree with a random subset of the main training set. A fourth option is to use different algorithms for generating each tree. There exist other more advanced approaches such as Dynamic Random Forest which create each tree as a complement of the previous tree [Bernard et al., 2012].

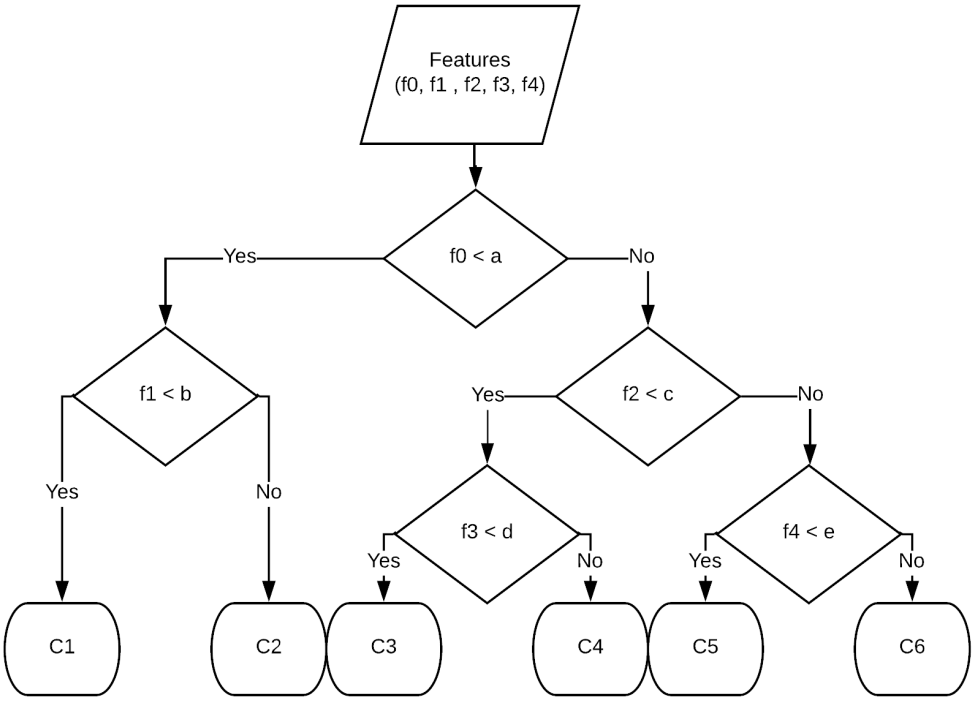


Figure 3.2: Example of Decision Tree

Chapter 4

Features

In this chapter, we will present the different features that can be extracted for each pixel of a satellite image.

4.1 Color Components

The first and obvious features are the color components of the pixel. These can be in several different color spaces. It is often useful to have either the classic red, green and blue (RGB) components or the hue, saturation and luminance (HSL) components. Other color spaces such as XYZ can also be used.

4.2 Gray-Level Co-occurrence Matrices

The Gray-Level Co-occurrence Matrices (GLCM) is measuring the gray-level similarity around a pixel [Haralick et al., 1973]. Each element M_{ij} of the matrix is the sum of the number of times that the discretized Luminance of a pixel p_{xy} $Lum(x, y) = i$ and $Lum(x + dx, y + dy) = j$ inside a defined window around each pixel, dx and dy being chosen displacement values. The window used in our implementation is a 15 by 15 window and the displacement vectors where $(0, 1)$, $(1, 0)$, $(1, 1)$, and $(-1, 1)$. From the GLCM, we can extract several features:

- **Contrast:** $K = \sum_{ij}(i - j)^2 M_{ij}$
- **Entropy:** $E = -\sum_{ij} log(M_{ij})M_{ij}$
- **Uniformity:** $U = \sum_{ij} M_{ij}^2$
- **Homogeneity:** $H = \sum_{ij} \frac{M_{ij}}{1+|i-j|}$
- **GLCM Mean:** $\mu = \sum_{ij} iM_{ij}$
- **GLCM Variance:** $\sigma^2 = \sum_{ij} M_{ij}(i - \mu)^2$

- **Correlation:** $C = \sum_{ij} M_{ij} \frac{(i-\mu)(j-\mu)}{\sigma^2}$
- **Shade:** $C = sgn(A)|A|^{1/3}$, where $A = \sum_{ij} \frac{i+j-2\mu^3 P_{ij}}{\sigma^3 2\sqrt{2(1+C)^3}}$

4.3 Histogram of Oriented Gradient

The Histogram of Oriented Gradient (HOG) is counting the occurrences of gradient orientation in a small window around a defined pixel. This means the the HOG essentially looks at the direction of edges in the texture around a pixel.

4.4 Local Binary Patterns

The Local Binary Patterns (LBP) vector compares a pixel with its neighbors to detect regular patterns. The LBP is calculated for a defined radius around the central pixel and for a defined number of points. The radius chosen for our implementation were 1, 2, 4, and 8 pixels with 8 points. As the LBP can be calculated separately for each color channel, it can be calculated on any color space.

4.5 Height, Normal and Slope

In the case that the height information is available for the classification process, it can be very useful to take it into account. Indeed, certain land cover types depend on the height and terrain slope at which they appear. For example, trees don't grow above 2000 meters of altitude.

Moreover, the height information can be derived to obtain the normal of the terrain. This information can also be useful, however in practice we will only use the slope of the terrain that we can get from the normal. Indeed, we are not interested by the orientation of the terrain as the overall land cover does not change much depending on the orientation. However, in the case we want to classify specific vegetation type, it would be useful to know the orientation from the sun.

Chapter 5

Implementation

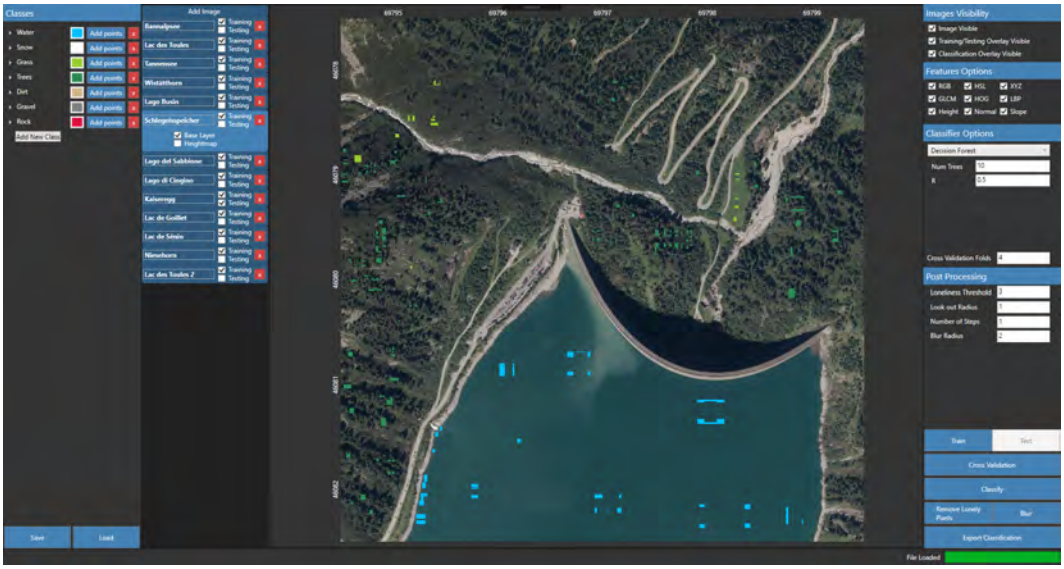


Figure 5.1: Our classification tool.

5.1 Requirements

Our classification tool is required to be be able to:

- load satellite[1] and height[2] data directly from ESRI servers.
- select features points on several different images for each different class.
- choose the parameters used for the classification algorithm as well as which features to use.
- classify each image and perform post processing if necessary.

[1] `http://services.arcgisonline.com/arcgis/rest/services/World_Imagery/MapServer/`
[2] `https://elevation3d.arcgis.com/arcgis/rest/services/WorldElevation3D/Terrain3D/ImageServer/`

- cross validate the classifiers.
- export the classification to be later use for rendering.
- batch process several images at once.

5.2 Framework and Libraries

The classification tool was written in C# using the Windows Presentation Foundation (WPF) toolkit by *Microsoft*. Two external libraries were used.

5.2.1 ALGLIB

ALGLIB is a cross-platform library designed for numerical analysis and data processing which is written in several languages, including C# [ALGLIB, 2018]. It is a versatile library which include several classification algorithms, optimization solvers, matrix operations, and some signal processing functions. This library was used for its implementation of the Decision Forest, which, although being well optimized, we improved by adding multi-threading capabilities.

5.2.2 Accord.Net

Accord.Net is a machine learning library which is written exclusively in C# [Framework, 2018]. It includes implementation for most supervised machine learning algorithm. Its implementation of Support Vector Machine comes with several kernels and is well optimized. However, while being only dedicated to machine learning, the implementation for Decision Forest is slower than the one included in the ALGLIB library. Nevertheless, this library contains a statistical module that is useful for the classifier validation.

5.3 Test and Validation

A common approach to test and validate a classification algorithm is to do a "cross-validation". To perform cross validation, the data set is partitioned randomly into a training set and a testing set. Then the classifier is trained using the first set and validated using the second set. This operation is done several times, called "fold", to have a result that does not depend on a particular training and testing set combination. Every following tests were done using 64 folds.

5.3.1 Testing Data

Our testing set is composed of thirteen different zones in the Alps of approximately 750m by 750m each. 20000 points of each class (Water, Snow, Grass, Trees, Soil, Gravel, and Rock) where manually classified, for a total of 140000 points. 10% of the points are used for training while the remaining ones are used for testing.

5.3.2 Accuracy Assessment

To tell how accurate a classifier is, we can use several values. The first and obvious one is the accuracy which measure how globally accurate the classifier is. This value is the percentage of points that were correctly classified. The precision and recall values can be used to measure the behavior of the classifier for each class. The precision measures the percentage of samples into a

Figure 5.2: Four of the thirteen zones used for testing.

predicted class which correct. On the other hand, the recall measures the percentage of samples into the actual class which were rightfully classified . For a more visual representation of these values, we can use a confusion matrix as in Table 5.1. While these values are great to measure the correctness of a classifier, it has been criticized because some correct predictions could have happened only by chance [Congalton, 1991]. Therefore, we can use the Cohen's Kappa value as well [Cohen, 1960]. A Kappa value of 1 or greater means that chance does not interfere in the result, while a value of 0 or lower means that the resulting accuracy is only based on luck.

		Actual Class				
		C_1	C_2	C_3	C_4	Precision
Predicted Class	C_1	$\#N_{11}$	$\#N_{12}$	$\#N_{13}$	$\#N_{14}$	$N_{11}/\sum N_{1j}$
	C_2	$\#N_{21}$	$\#N_{22}$	$\#N_{23}$	$\#N_{24}$	$N_{22}/\sum N_{2j}$
	C_3	$\#N_{31}$	$\#N_{32}$	$\#N_{33}$	$\#N_{34}$	$N_{33}/\sum N_{3j}$
	C_4	$\#N_{41}$	$\#N_{42}$	$\#N_{43}$	$\#N_{44}$	$N_{44}/\sum N_{4j}$
	Recall	$N_{11}/\sum N_{i1}$	$N_{22}/\sum N_{i2}$	$N_{33}/\sum N_{i3}$	$N_{44}/\sum N_{i4}$	

Table 5.1: Confusion Matrix Example

- **Accuracy:** $A = \frac{\sum_i N_{ii}}{N}$
- **Precision:** $P_i = \frac{N_{ii}}{\sum_j N_{ij}}$
- **Recall:** $R_j = \frac{N_{jj}}{\sum_i N_{ij}}$
- **Kappa:** $\kappa = \frac{A-p_c}{1-p_c}$ where: A is the accuracy and $p_c = \frac{1}{N^2}\sum_{ij}\left(\frac{N_{ij}N_{ji}}{2}\right)^2$ is the expected accuracy.

Chapter 6

Building our Classifier

6.1 Algorithm Comparison

6.1.1 Comparison in the Literature

SVM are mostly use for land classification when multiple bands of the color spectrum are available (i.e. infra-red, visible, ultra-violet, ...). In Huang et al. [2002], the authors compared SVM against Decision Tree and Neural Network for land classification using several bands combinations. They conclude that SVM outperforms both other algorithms when using 7 bands while the Neural Network perform slightly better when using only 3 bands. In another study by Otukei and Blaschke [2010], they compared SVM against Maximum Likelihood and Decision Trees. They conclude, contrary to Huang et al., that the Decision Tree outperforms both other algorithms. This difference can be explained by the fact that the land cover that were classified were not the same in both studies.

However, in these studies, they are using multi-band data, which we are not able to use. Indeed, the only data that we have directly available are the satellite image and the surface height information. In Argudo et al. [2017], the author compared several classification algorithms while using only these data plus the near-infrared band. The results of their research demonstrate that the random forest outperform in other algorithms, including SVM. Indeed, in addition to a better overall accuracy, the training and inferring time are also the best possible, with only 6min to train a forest with an accuracy of 94.80% while the best SVM with an accuracy of 94.68% needed more than 13 hours to train.

6.1.2 Support Vector Machine

The parameters for the Support Vector Machine depend on which kernel is used. For the linear case, only the complexity needs to be set. The Accord.net library provide an easy way to estimate the best complexity value possible for the task. This is done inside the framework by using a heuristic rule. Using the found value of 0.47, the linear kernel results in an accuracy of 67.24% with a Kappa value of 0.845.

For the Polynomial case, we need to set the polynomial degree to use as well as the complexity. Once again, the complexity is automatically estimated, this time to a value of 0.09. As seen in Figure 6.1, the result with the best accuracy is the one with a polynomial degree of 3, which results in an accuracy of 69.18% with a Kappa of 0.99. Therefore, the polynomial kernel performed slightly better than the linear one.

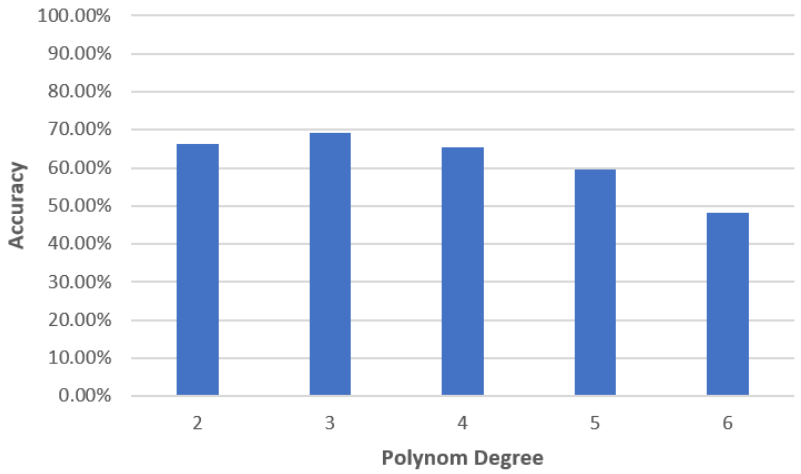


Figure 6.1: Accuracy changes according to the polynomial degree.

Finally, for the Gaussian kernel, we need to set the sigma value in addition to the complexity. This time, the complexity is set to 1. As we can see in Figure 6.2, the best sigma value for the SVM with Gaussian kernel is 0.35, with an accuracy of 82.36% and a Kappa value of 0.97. In conclusion, the Gaussian kernel performed better than the linear and polynomial ones.

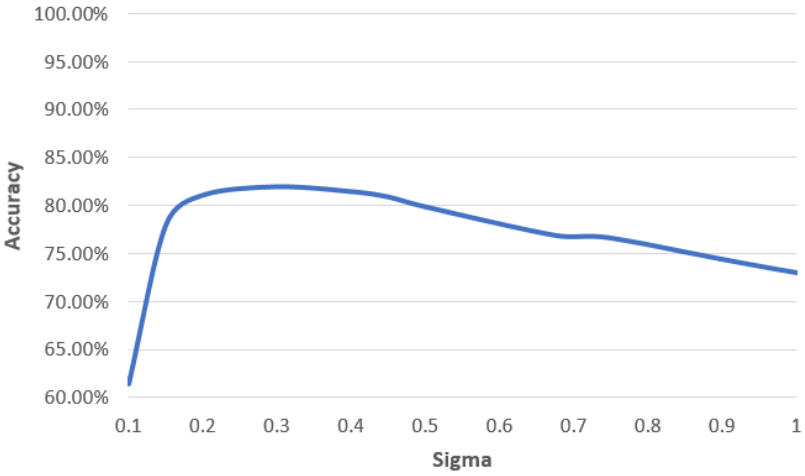


Figure 6.2: Accuracy changes according to the sigma value for a Gaussian kernel.

6.1.3 Decision Forest

In the ALGLIB implementation, there are two parameters that can be chosen for the Decision Forest. The first one is the number of trees in the forest and the second one is the proportion of samples that are effectively used for the training. The latter parameter will be fixed to 1 as we already partition the data set into a training and testing set. To choose the best number of trees in our forest, we need to test several possibilities with our data set. Too few trees will lead to under-fitting, while too many will lead to over-fitting. By looking at Figure 6.3, we can see

that between 20 and 100 trees, the overall accuracy doesn't change much. In our case, the best accuracy is 98.18% at 80 trees. One other thing to consider when changing the number of trees, is that the construction time and memory usage increase linearly according to the number of trees. Indeed, while it only takes 14 seconds[1] and 28 MB of memory to compute a forest with ten trees, it will take ten time more time and memory to compute a forest of a hundred trees.

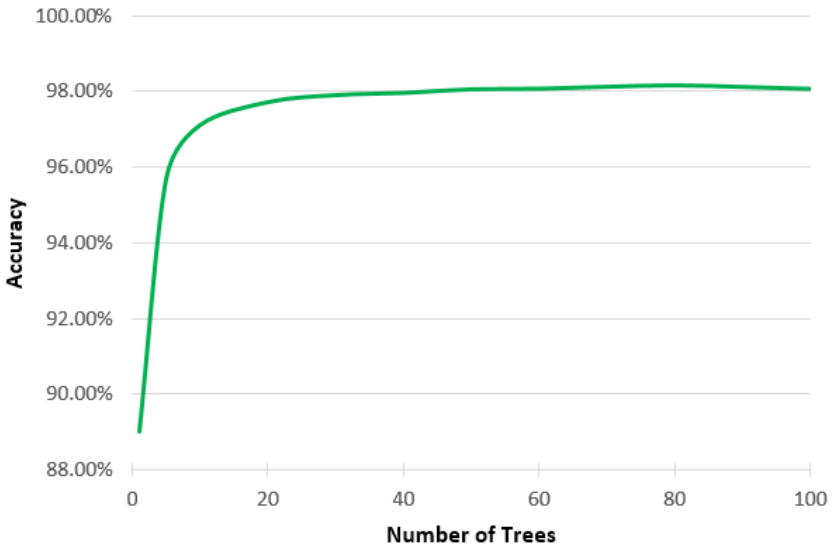


Figure 6.3: Accuracy changes according to the number of trees in the forest.

6.1.4 Algorithm Choice

As we can see in the two previous section, the Decision Tree performed much better than the any SVM we tested. Indeed, with an impressive 98.18% of accuracy, it easily outperformed the 82.36% accuracy of the SVM with a Gaussian kernel. Therefore, from now on, we will only use the Decision Forest and discard the SVM.

6.2 Feature Selection

While we saw in Chapter 4 that there exist numerous different features that we can use, it is not necessarily a good idea to use them all at once. Indeed, some features can in certain case give worse results than without them. To test that, we can simply test our algorithm by taking each feature out to see how much does each one impact the overall accuracy. As we can see in Figure 6.4, the RGB and XYZ features impact the classifier negatively. Indeed, the overall accuracy increase, however not by much. On the other hand, when not using the HSL, GLCM, LBP, and height features, we are losing accuracy by 0.4%, 0.8%, 0.2%, and 1.4% respectively. Finally, the HOG and Slope features don't impact the accuracy by much, only 0.02% and 0.05% respectively.

Therefore, in the final classifier, we will use every feature except the RGB and XYZ color channel. This will result in a final overall accuracy of 98.76% with a Kappa of 0.99. The final confusion matrix for this composition of features can be seen in Table 6.1

[1] Computation done with a Intel® Core ™ i7-2600 CPU @ 3.40 GHz (4 cores / 8 threads)

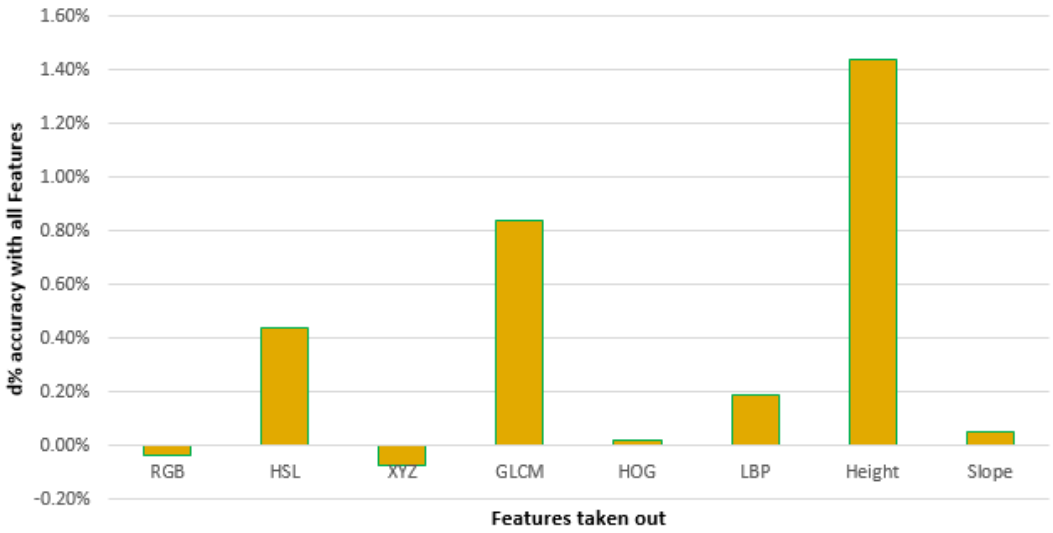


Figure 6.4: Accuracy changes when taking a feature out. A negative percentage means that the classifier performed better without the feature.

		Actual Class							
		Water	Snow	Grass	Trees	Soil	Gravel	Rock	Precision
Predicted Class	Water	1142762	483	486	73	259	133	1414	99.75%
	Snow	51	1146573	69	1	288	8665	1063	99.12%
	Grass	4813	39	1119916	7729	3350	843	1312	98.41%
	Trees	1538	48	24660	1143532	772	523	1745	97.5%
	Soil	792	310	3603	155	1143135	4199	3729	98.89%
	Gravel	640	2545	250	1	1267	1131210	6056	99.06%
	Rock	1404	2002	3016	509	2929	6427	11366081	98.59%
	Recall	99.2%	99.53%	97.21%	99.26%	99.23%	98.2%	98.67%	

Table 6.1: Final Confusion Matrix

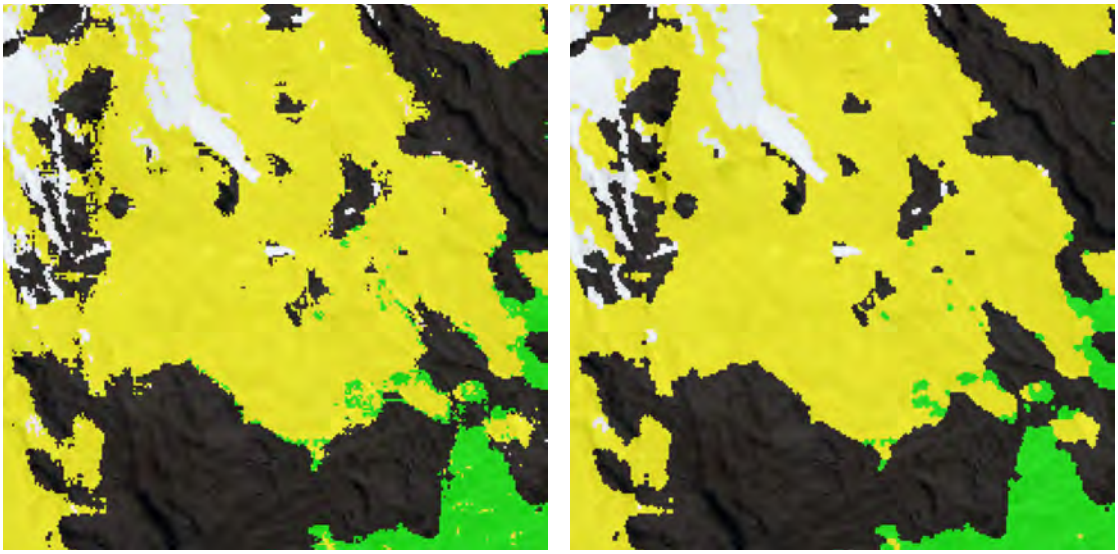
6.3 Classification Post-Processing

While the final classifier performs extremely well, there are still some pixels that are miss classified, which produce noise in the final result. To counter that, we can apply two different post processing steps: filter "lonely pixels" and blur the edges of the classes.

6.3.1 Filtering "lonely pixels"

At a precision of 1.2m per pixels, it is unlikely that a different land cover will appears on a single pixel surrounded by a different cover. Therefore, we can filter those lonely pixels to be able to

reduce the noise in the classification. This filtering can be done individually for each class and with different "loneliness threshold". Indeed, the size of land cover patches can be different for each class. For example, we can expect a patch of grass of 3 by 3 pixels, but it is unlikely to find a patch of water that has the same size. In our final classifier, we used values of 7 by 7 and group of at least 30 pixels for water, 5 by 5 and a group of at least 15 pixels for the forest, and 3 by 3 with a group of at least 3 pixels for the other ground covers.

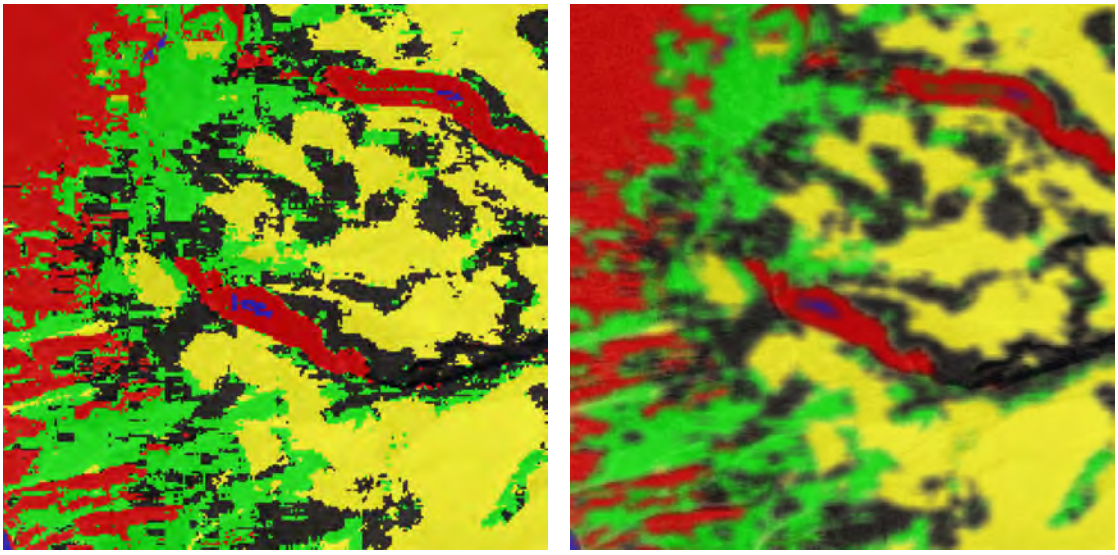


(a) (b)

Figure 6.5: Comparison before (left) and after (right) lonely pixel filtering.

6.3.2 Blurring

A simple Gaussian blur can also help to reduce the noise in the classification, especially at the edge of two different land cover. The blurring process will make the change of land cover smoother and will thus be also helpful when using the classification for rendering. Indeed, while each pixel belonged to a unique class before the blur, they can belong to several classes with different weight. In our final classifications, the Gaussian blur was applied on 11 by 11 windows.



(a) (b)

Figure 6.6: Comparison before (left) and after (right) Gaussian blur.

Chapter 7

Conclusion

In this first part, we looked at the two most used land classification techniques as well as how they performed in different research. We saw that, according to the literature, Random Forest was the most suitable algorithm for our problem as it performed better than SVM when using only satellite images and height data. Indeed, in addition to a higher accuracy, it is also quicker to train and classify new points.

In Chapter 4, we looked at the features that we can extract and use for satellite images. Except from the RGB color values, we can use other color space such as HSL or XYZ. Beside the color information, we can also use other more complex features that consider the surrounding of each pixels, such as GLCM, HOG and LBP. We also looked at how the height data can be used.

Finally, we saw at how we implemented and tested our classification tool. We saw how the classifier was tested and validated. We compare the performance of the Decision Forest and the SVM with different parameters. We concluded that, as expected, the Decision Forest performed better than SVM.

Moreover, the importance of each feature was investigated, and we saw that using all features described in Chapter 4 was not optimal. Indeed, we saw that the RGB and XYZ features impacted the classifier slightly negatively, which is why those were not included in our final classifier. At the same time, we saw that the height, GLCM and HSL features were the three most important one to have for a successful classification. Figure 7.1 show a classification example done with our final Random Forest using 80 trees and an accuracy of 98.76%. Using this classifier, the classification and post-processing of one tile is approximately 20 seconds on a Intel®Core ™i7-2600 CPU @ 3.40 GHz (4 cores / 8 threads). We can see in Table 7.1 below, the time required to classify entire regions at level 17 using the same computer. These times can be greatly reduced by using multiple computers with more powerful CPUs, or even use GPUs. Another solution to increase the speed of the classification would be to only classify lower zoom level. As the number of tile is divided by four at each zoom level decreased, it would be possible to classify the five continents in less than a year at zoom 12, instead of 934 years at level 17.

Region	**Area** [km^2]	**Estimated Tiles**	**Time** [years]
Switzerland	41'285	437'472	0.28
USA	9'833'517	104'199'696	67.7
Europe	10'180'000	107'871'162	70.1
Africa	30'415'873	322'298'188	209
Asia	43'810'852	464'236'493	302
Oceania	8'525'989	90'344'631	123
North America	24'709'000	261'825'986	170
South America	17'840'000	189'039'443	123
Total	135'481'714	1'435'615'903	934

Table 7.1: Time to classify entire region at zoom level 17 on the same computer used for this thesis

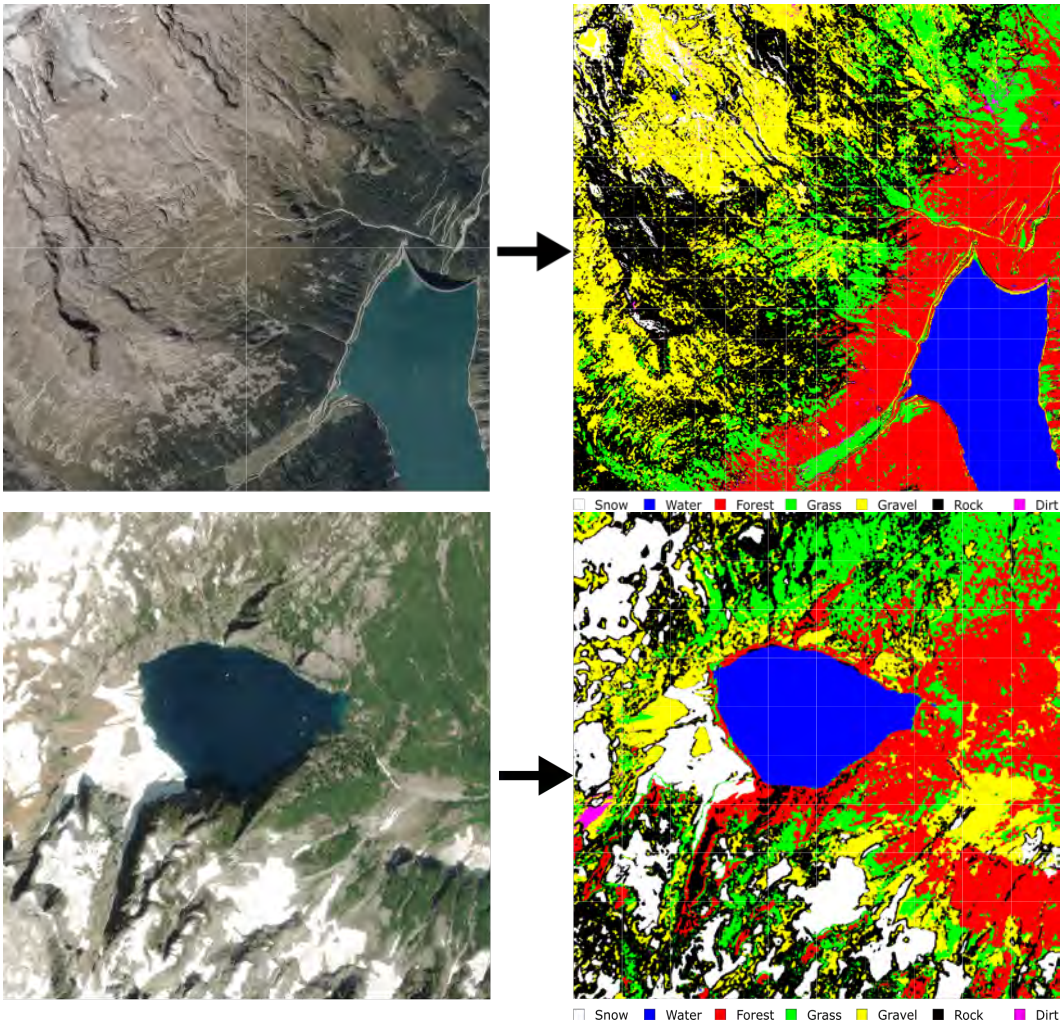


Figure 7.1: Two examples of satellite image classification. Top is in the Swiss Alps. Bottom is in the Rocky Mountains in USA.

Part II

Rendering

Chapter 8

Introduction

In this second part, we are going to look how we can enhance the terrain rendering using the classification data extracted from the first part. We will explore how we can use the classification to render specific materials for each land cover. While the initial goal of this thesis was to do the rendering part in WebGL using the ArcGIS API for Javascript, we decided to prototype the realistic terrain rendering in Unity using a HLSL (DirectX) shader. These was done for two main reasons. The first one being the ease to prototype and iterate quickly between different techniques. The second reason was that we could experiment with the best quality possible without compromise. However, the ability to easily port the result to WebGL was kept in mind throughout the project.

As the goal is to have a realistic and highly detailed terrain surface rendering by adding details that are not available in the satellite imagery, several components need to be develops. The major ones are: realistic shading, scalable materials guided by the classification, real-time normal and parallax mapping for increased details, and real-time water simulation and rendering.

In the first chapter we will present the basics of physically based rendering (PBR) and how it can be applied for real-time use. We will especially look at bidirectional reflectance distribution functions and global illumination.

In the second chapter, we will look at how the ground materials can be rendered using multi-scale texturing. We will also focus on the transition between two, or more, different materials. We will finish this chapter by looking at parallax displacements.

In the third chapter of this part, we will focus on water rendering. Firstly, we will look at how we can simulate waves using Gerstner waves. Finally, we will look at the special BSDF the water is using.

Chapter 9

Physically Based Rendering

9.1 Introduction

Contrary to older shading models, such as Phong and Blinn-Phong, which were created from observations, the PBR shading models are based on real physical laws of light interaction. Indeed, PBR uses physical particularities such as the Fresnel law and energy conservation. The Fresnel law states that the reflectively of a surface increase with low viewing angles. In addition, to physical laws, PBR uses the microfacet theory to be able to reproduce smooth or rough surface. Indeed, this theory aims to simulate the micro-bumps (i.e. roughness) on a material that would be too costly to model or even reproduce with normal maps.

9.2 Variables

v	view direction
l	light direction
n	normal
$h = normalize(v + l)$	half direction
$\theta_v, \theta_l, \theta_h$	angle between the normal and the view, light and half direction respectively
θ_d	angle between the half and view direction
R	roughness
C_α	camera field of view in degrees
C_h	camera rendering target height in pixel
Z	depth; distance to the point

Table 9.1: variables for shading

9.3 Bidirectional Reflectance Distribution Function

The Bidirectional Reflectance Distribution Functions (BRDFs) are functions that defined how the light interacts with an opaque surface. These functions use the light and the view directions to calculate the shading of the surface. As seen in 9.1, the general form of an microfacet BRDF can be expressed as an addition of a diffuse and a specular term:

$$\mathbf{f}(\mathbf{l}, \mathbf{v}) = \mathbf{f_d}(\mathbf{l}, \mathbf{v}) + \mathbf{f_s}(\mathbf{l}, \mathbf{v})$$

A physically based BRDF follows three principal characteristics:

- Positive: $\mathbf{f}(\mathbf{l}, \mathbf{v}) \geq 0$
- Helmholtz reciprocity[1]: $\mathbf{f}(\mathbf{l}, \mathbf{v}) = \mathbf{f}(\mathbf{v}, \mathbf{l})$
- Energy conservation: $\int_\Omega \mathbf{f}(\mathbf{l}, \mathbf{v}) \cos\theta_v d\mathbf{v} \leq 1$

In 2012, the *Walt Disney Animation Studio* developed their Principled BSDF[2] that aimed to be physically correct and being artists-friendly [Burley and Studios, 2012]. Indeed, the complete BSDF enable artists to set up the diffuse color, metalness, roughness, subsurface scattering, sheen, and clearcoat using the same shader while keeping the number of exposed parameters to a bare minimum. In our case, as we only use the diffuse and specular part, we solely need to set the base color and roughness to achieve realistic shading.

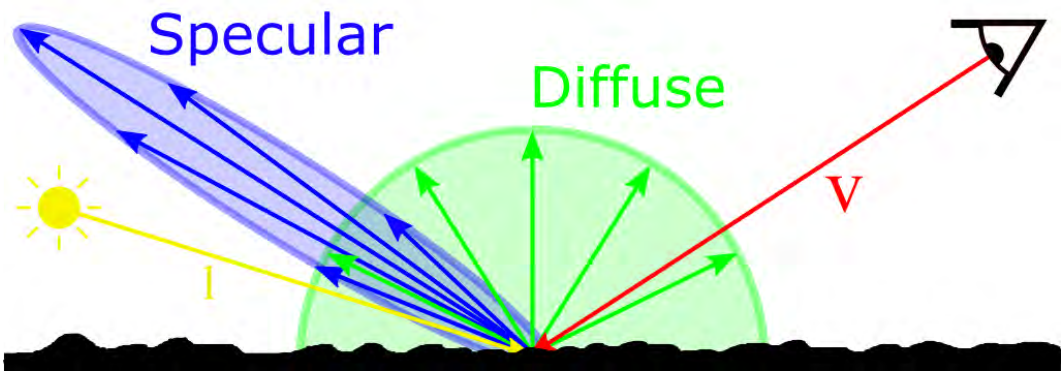


Figure 9.1: Separation of Diffuse and Specular term of a BRDF.

9.3.1 Diffuse BRDF

The Disney Principled Diffuse BRDF is described as follow:

$$\mathbf{f_d}(\mathbf{l}, \mathbf{v}) = \frac{baseColor}{\pi} \left(1 + (F_{D90} - 1)(1 - \cos\theta_l)^5\right) \left(1 + (F_{D90} - 1)(1 - \cos\theta_v)^5\right)$$

where:

$$F_{D90} = 0.5 + 2 \cos^2\theta_d \, R$$

We can see in Figure 9.2 a comparison between the real-time rendering in Unity and the offline render in Blender Cycle. The renders are done using a sun/directional lamp in both software with the same intensity and color. While the results are not exactly similar, the difference is barely visible. The most probable reason as to why the results are slightly divergent is that the color management in Blender and Unity are different.

[1] Hapke [2012]

[2] Bidirectional Scattering Distribution Function: a more versatile type of BRDF that take into account scattering inside the surface. Useful for material such as glass and skin.

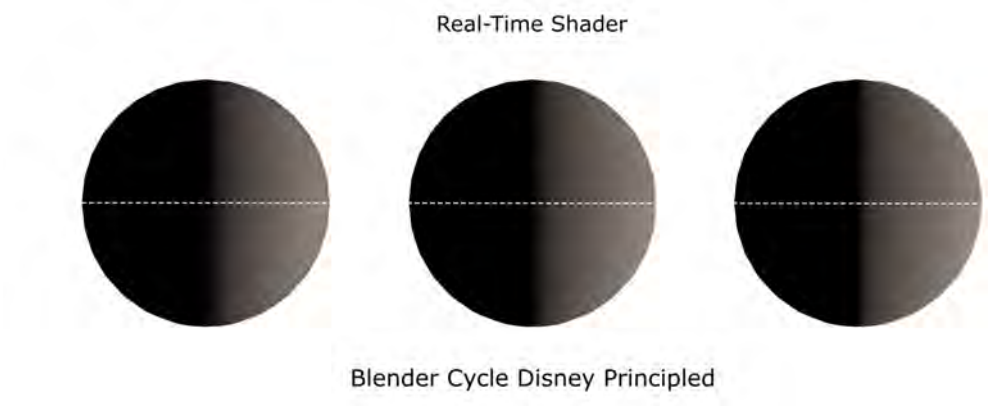


Figure 9.2: Comparison of Diffuse term between our implementation (top half) and the implementation included in Blender Cycle (bottom half) with roughness value of 0.0 (left), 0.5 (center), and 1.0 (right).

9.3.2 Specular BRDF

The specular part of the microfacet BRDF can be expressed as:

$$\mathbf{f_s}(\mathbf{l}, \mathbf{v}) = \frac{D(\theta_h)F(\theta_d)G(\theta_l, \theta_v)}{4\cos\theta_l \cos\theta_v}$$

Where $F(\theta_d)$ is the Fresnel factor, $D(\theta_h)$ is the distribution, and $G(\theta_l, \theta_v)$ is the shadowing function. As for the Disney Principled BSDF, a GGX distribution was used, which defined D and G for a roughness R as:

$$D(\theta_h) = \left(\frac{R^2}{\cos^2\theta_h\ (R^2-1)+1}\right)^2$$

$$G(\theta_l, \theta_v) = G_1(\theta_l)G_1(\theta_v)$$

with

$$G_1(\theta) = \frac{2}{1+\sqrt{1+R^2\tan^2\theta}}$$

The Fresnel factor use the Schlick Fresnel approximation which is defined as

$$F(\theta) = F_0 + (1-F_0)\ (1-\cos\theta)^5$$

with F_0 being the specular reflection at normal incidence.

In Figure 9.3, we can see a comparison of the specular term between the render in Unity and the one done with Blender. As the diffuse part, there is a slight difference which is visible with high roughness. This difference can be caused by the same color management issue. However, in the case of specular reflection, it is most likely the way the sun light is handled in Unity and Blender. Indeed, in Unity, the directional light is modeled as constant directional lamp with no effective size for the light source. On the other hand, Blender Cycle uses a disc as light source, which makes the specular reflection spot larger.

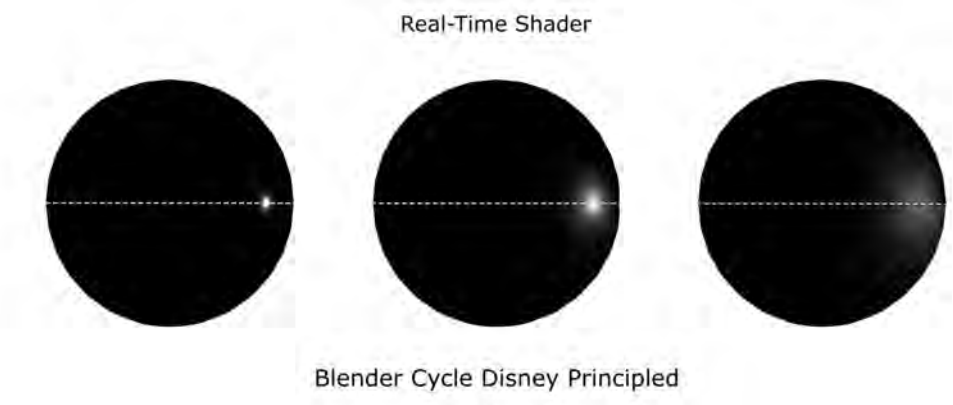
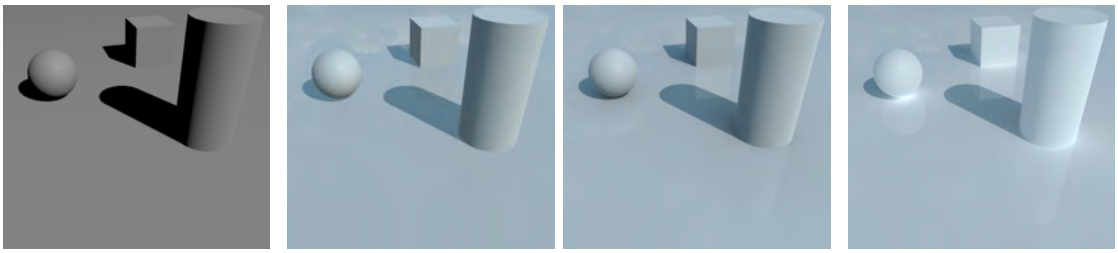


Figure 9.3: Comparison of Specular term between our implementation (top half) and the implementation included in Blender Cycle (bottom half) with roughness value of 0.1 (left), 0.3 (center), and 0.5 (right).

9.4 Global Illumination

In the real world, objects are not only lit by direct light source, but also by the light scattered in the atmosphere and the one which is reflected by surrounding objects. This is called global illumination. In offline rendering, this effect can easily be simulated by ray tracing. However, in the case of real-time rendering, simulating each ray of light is not possible with current hardware. On the other hand, our goal is to render terrain, therefore, there are not any other objects than the landscape itself. This is good for us because it means that the secondary illumination is mainly coming from the sky, and only a small portion from the ground itself.

Therefore, we can fake global illumination by rendering an environment map of the surrounding, and us it as illumination. The map is easily created by using a reflection probe in Unity. This creates a cube map which sides are 512 by 512 pixels. As only one cube map is rendered for the whole scene, the illumination will not be perfect, and it will not be possible to have results as in Figure 9.4 (d) which calculate a true global illumination with 128 ray bounces. However, we can still achieve results close to the offline render with only one light bounce allowed as we can see in Figure 9.4 (b) and (c).



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

Figure 9.4: Render without (a) and with (b) global illumination compared to the reference render done with Blender Cycle allowing one ray bounce (c) and 128 ray bounces (d).

As it was done for the BRDF, we can separate the diffuse global illumination and the specular

reflection in two different parts.

9.4.1 Diffuse

To simulate the diffuse component, we will take the same approach as offline rendering: sampling rays in a hemisphere. We will use a standard cosine importance sampling strategy. Depending on the quality setting chosen, 8 to 64 outgoing directions will be sampled. These outgoing vectors are used to sample pixels on the environment cube map.

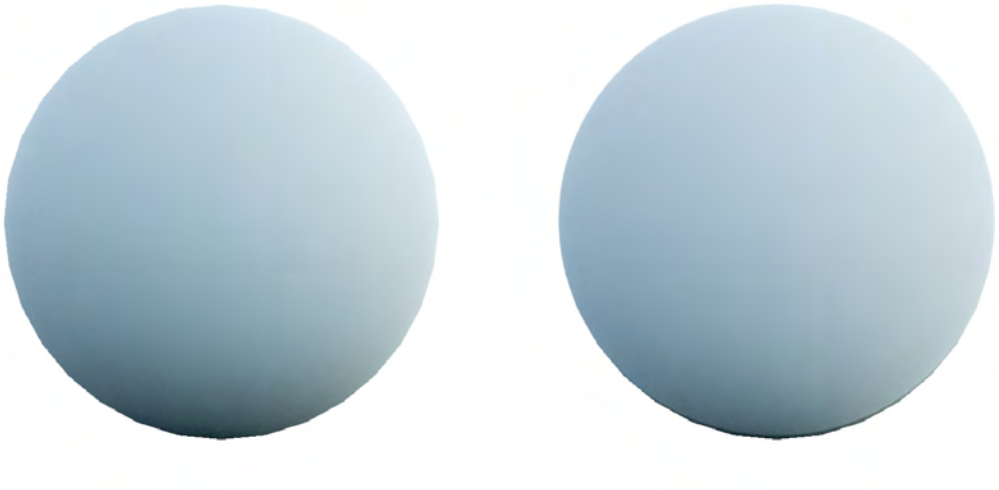


Figure 9.5: Comparison of diffuse component between real-time (left) and the Blender Cycle render reference (right). The exposure was increase by 50% for clarity.

9.4.2 Specular

Contrary to the diffuse component, the specular component does not need to actually sample several different outgoing directions. Indeed, as seen in Figure 9.1, the specular rays are approximately all going toward the same direction. This help us reducing the complexity of the specular calculation by only sampling one direction, and thus only sampling the cube map once. Indeed, instead of sampling multiples ray that will most likely go toward the same general direction, we can simply sample one direction and use a blurred version of the cube map to get rough reflections. To get a blurred cube map, we will sample different mipmap level depending on the roughness:

$$MipMapLevel = \sqrt{R}\ MipMapCount$$

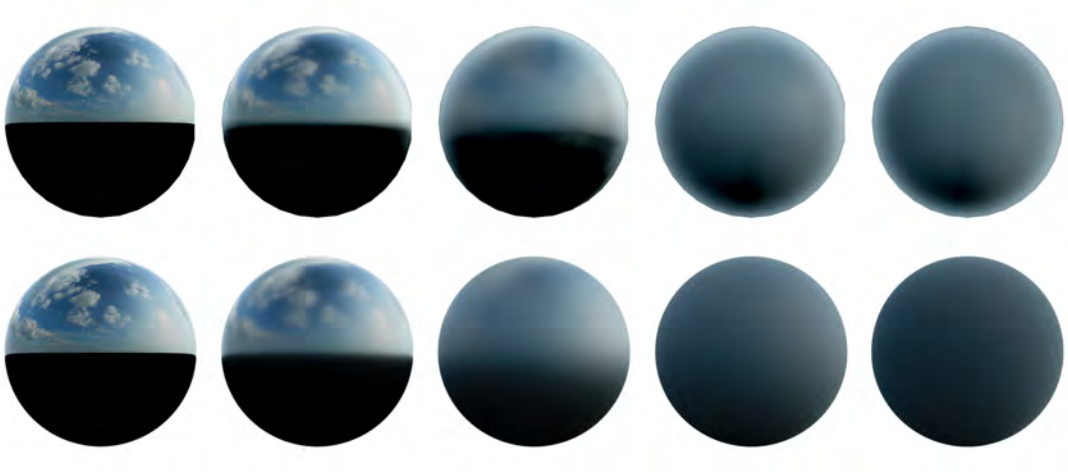


Figure 9.6: Comparison of specular component with roughness 0, 0.2, 0.5, 0.8, 1.0: Real-time render (top) and reference Blender Cycle render (bottom). The exposure was increase by 50% for clarity.

9.4.3 Global illumination for landscape rendering

In Figure 9.7, we can see the effect of global illumination for the landscape rendering. We can see that the lighting is much more realistic with than without.

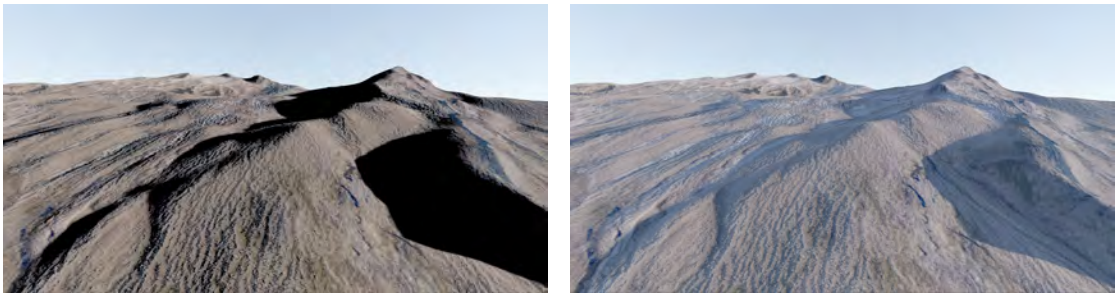


Figure 9.7: Comparison of the landscape rendering without (left) and with (right) global illumination.

Chapter 10

Ground Materials

10.1 Introduction

As the main goal of this thesis is to render landscape from far away to close-up, we need a way to have small scale details as well as large ones. This can be achieved by using materials with multiple layers of details. Multiple techniques ranging from full procedural to texture-based can be used for such effect. In this chapter, we will present a method to achieve multi-scale materials with a texture approach.

While the initial idea was to use a full procedural texturing strategy, it was quickly dismissed for a more classical texturing approach. Indeed, while it is possible to create realistic looking materials using procedural techniques, it is computationally too expensive to create all the material needed in real-time. The few tests we made required more than 100 co-sinus to create a realistic looking texture with a single color channel. Given the number of different materials we want to render, it would have be impracticable to it in real-time.

All ground materials (snow, rock, gravel, soil, grass) are using three levels of texture that seamlessly blend to achieve multi-scales rendering. The first layer, layer 0, is based on the satellite image. On the other hand, the second and third layers, layer 1 and 2, are based on highly detailed ground textures. All these materials use the BRDF as described in chapter 9.

10.2 Material Parameters

As seen in the previous chapter, the BRDF itself needs two values, the base color and the roughness, to achieve a realistic shading. Therefore, these two values will be specific for each material. Moreover, a third value will be set to achieve a bump effect: the normal.

In addition to these three parameters, a fourth one, the height, is also require when dealing with parallax and material blending, which we will discuss later on. Each parameter is set up by various textures and values depending on the level of details.

10.3 Textures Level of Details

As we are dealing with large landscape, we need to use different level of detail (LOD) for the texturing sampling depending on the depth. Indeed, without different LOD for close and long distances, aliasing will appear on the textures. We can solve that by using texture mipmap. To choose which mipmap level to sample, we need to consider the depth and the orientation of the

surface. Therefore, the LOD level used for a surface at distance Z, with $v \cdot n = \cos\theta_V$ and with a UV multiplier μ can be found using the following formula:

$$LOD_{tex} = log_2\left(\mu \times Z \times \Gamma \times \frac{1}{cos\theta_v} \times \left(\frac{1}{tan\theta_v - \Gamma} + \frac{1}{tan\theta_v + \Gamma}\right)\right)$$

$$\Gamma = \frac{tan\left(\frac{C_\alpha}{2}\right)}{C_h}$$

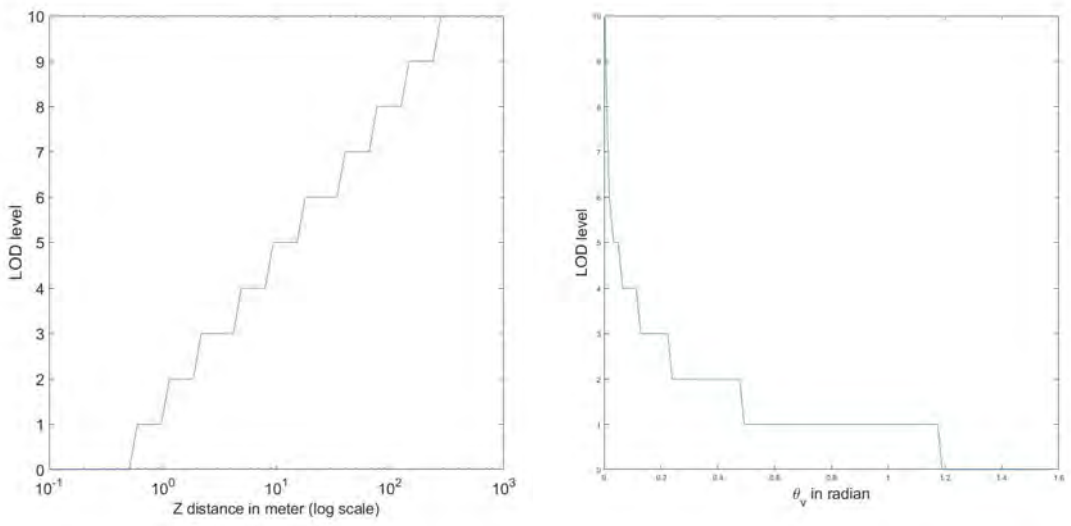


Figure 10.1: LOD level depending on varying Z with fixed $\theta_v = \frac{\pi}{4}$ (left) and varying θ_v with fixed $Z = 1$ (right). $\mu = 1000$, $C_\alpha = 45°$, $C_h = 1080$.

10.4 Color Transfer

With the multi-scale texturing approach, a problem arises: the average color from the 3 layers can change drastically depending on the textures used, and therefore, new colors could pop out when zooming. To minimize this issue, we can transfer the color from layer to layer in order to have a consistent tint across all scales. To do that we will first compute the average color of a texture from the highest layer, which can be found by sampling its highest mipmap level, and subtract if from the color of the texture at the correct LOD level. Then we will simply add the result to the color from the lowest layer. The resulting color, or texture, will have the same tint as the lowest layer, but will keep the details from the highest layer.

$$Color = Color_{L0} + (Color_{L1} - AvgColor_{L1})$$

Figure 10.2: Example of transferring the left color from the left texture to the detailed texture in the center.

10.5 Layer 0

The first layer is identical for all ground materials. The base color is simply the satellite image. The roughness is set to a constant value of 1.0. Concerning the normals, they can be set to a constant $n = (0, 0, 1)$. However, we can have some extra details by derivating the satellite images to fake normals.

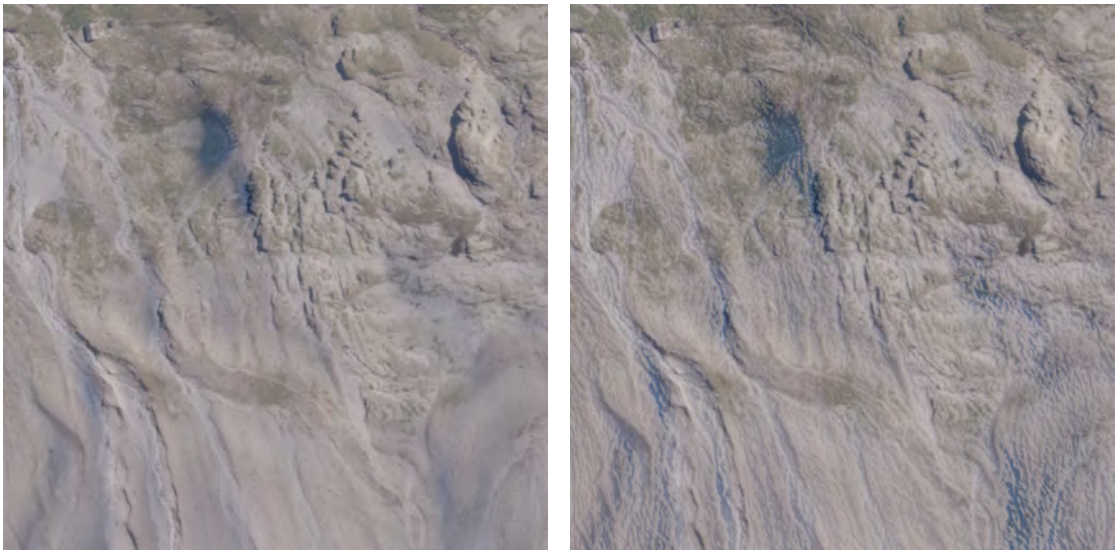


Figure 10.3: Comparison of level 0 render with (left) and without (right) normals.

10.6 Layer 1 and 2

Contrary to the first layer, the two other layers are defined separately for each ground material. These layers are mostly composed of realistic textures. The textures for the rock, snow, gravel, and soil are 3D scanned textures from Textures.com. The textures for the grass and forest were rendered in Blender Cycle, using realistic grass strands and trees. In addition to the textures,

several other parameters such as UV multipliers, roughness modifier or normal strength can be set.

10.6.1 Gravel and Soil

Gravel and soil use the same textures for the second layer. This is motivated by the fact that while being two largely different surface types, the large-scale shapes behave in a similar fashion. Moreover, using only one set of textures for two materials reduce the total number of textures needed and thus reducing the required video memory.

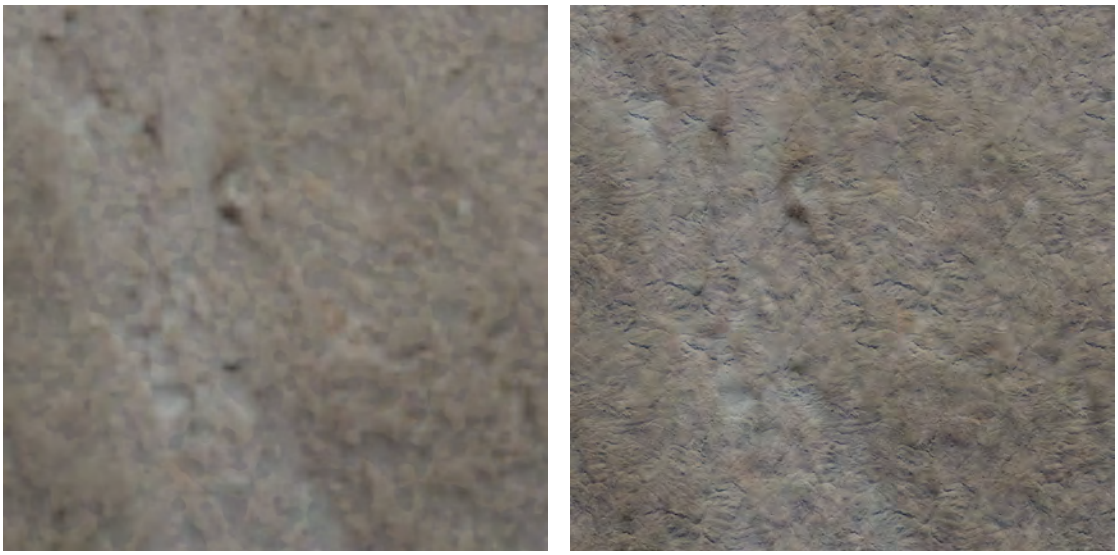


Figure 10.4: Render comparison of level 0 (left) and level 1 (right) at 100m for gravel and soil material.

Figure 10.5: Render comparison of level 1 (left) and level 2 gravel (center) and level 2 soil (right) at 2.5m.

10.6.2 Rock

Figure 10.6: Render comparison of level 0 (left) and level 1 (right) at 100m for rock material.

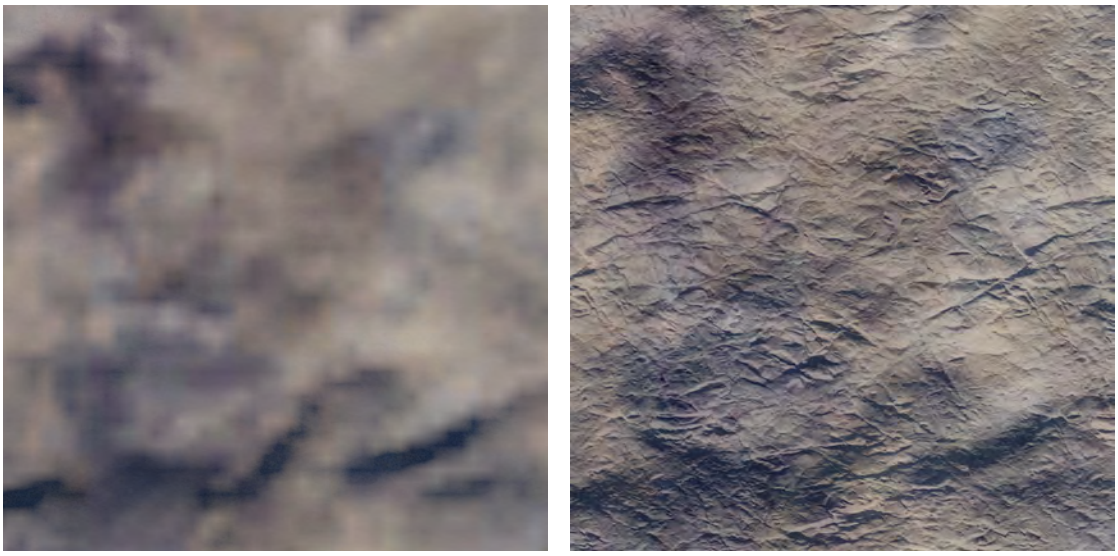


Figure 10.7: Render comparison of level 1 (left) and level 2 (right) at 2.5m for rock material.

10.6.3 Snow

Because the snow is mainly white, there is no base color texture for the level 1 and 2, instead a constant white color is chosen. The roughness was also set to a constant number, as the roughness textures add almost no roughness variation.

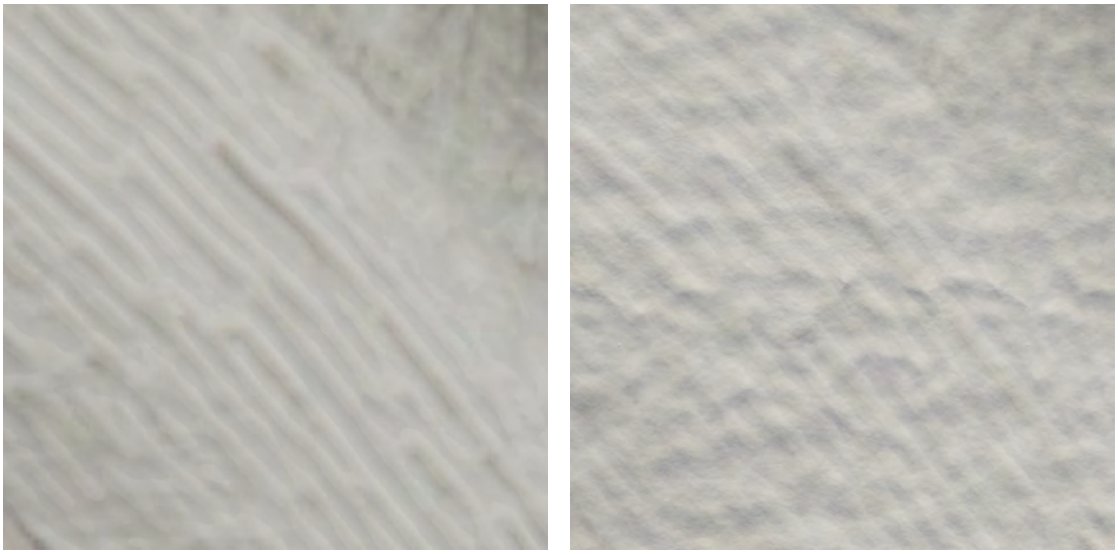


Figure 10.8: Render comparison of level 0 (left) and level 1 (right) at 100m for snow material.

Figure 10.9: Render comparison of level 1 (left) and level 2 (right) at 2.5m for snow material.

10.6.4 Grass

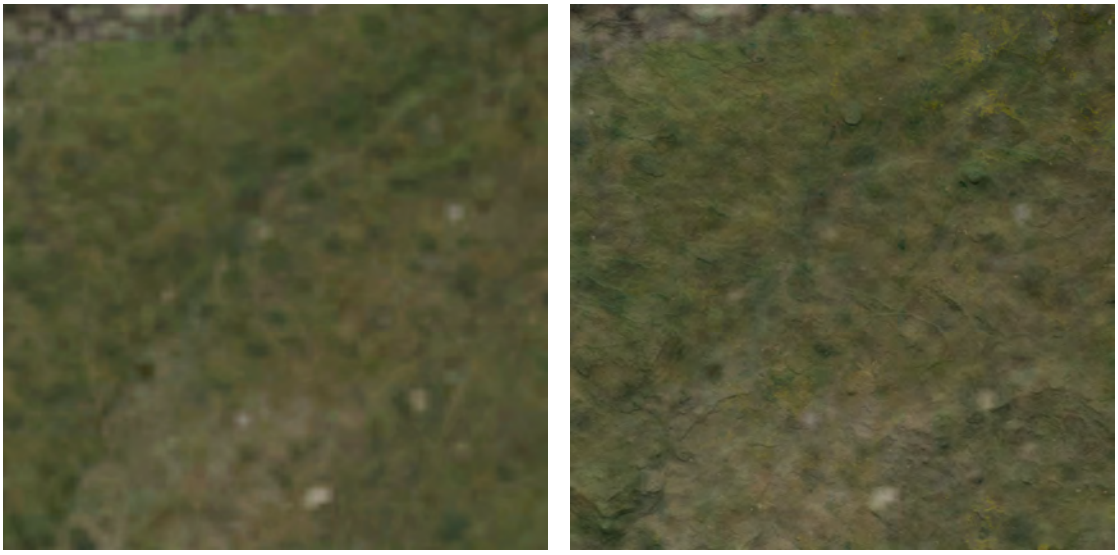


Figure 10.10: Render comparison of level 0 (left) and level 1 (right) at 100m for grass material.

Figure 10.11: Render comparison of level 1 (left) and level 2 (right) at 2.5m for grass material.

10.6.5 Forest

The forest material does not use level 2 textures. Indeed, the choice was made to only focus on the large-scale part of the forest during this project. This is why only the forest canopy is visible. However, an extension could be made to see the forest ground, below the trees, when close to it.

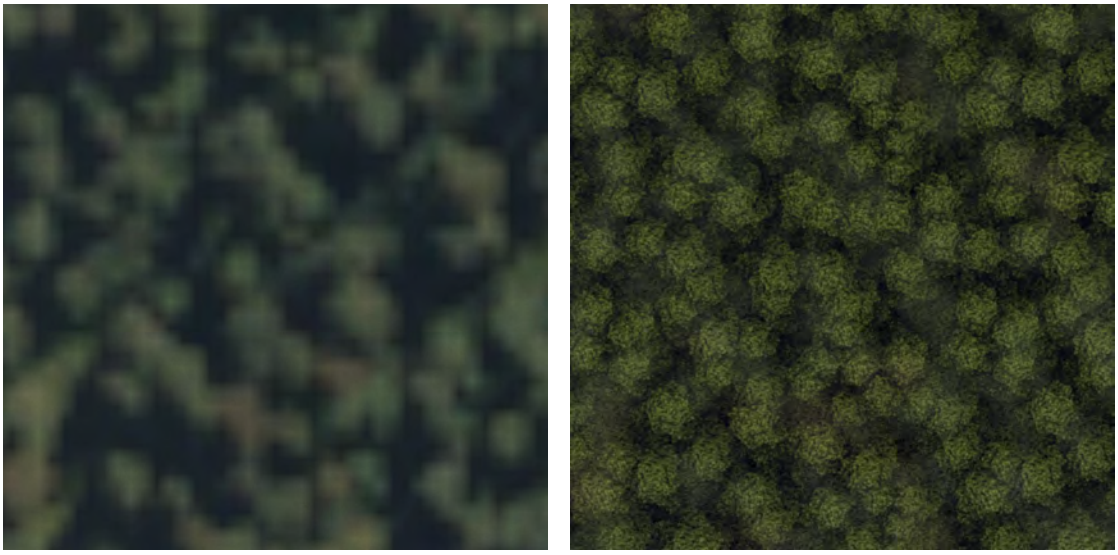


Figure 10.12: Render comparison of level 0 (left) and level 1 (right) at 100m for forest material.

10.7 Material Blending

As the terrain is composed of multiple ground types, several kinds of materials interact with each other. This interaction comes from the blur that was done at the end of the classification process. In this section, we will take a look at two different kinds of blending: interpolated blending and height-based blending.

10.7.1 Interpolated Blending

Interpolated blending is the easiest and most common type of blending. It allows to smoothly change from one material to another by mixing them. Each material m_k as a define weight w_k attributed to it, which is extracted from the classification images that we obtained in Part 1. The weights fall-off can be modified to have a linear or sigmoid results for example. The resulting material from an interpolated blending can be calculated as:

$$M = \sum_{k=0}^{N} w_k m_k, \text{ where } \sum_{k=0}^{N} w_k = 1$$

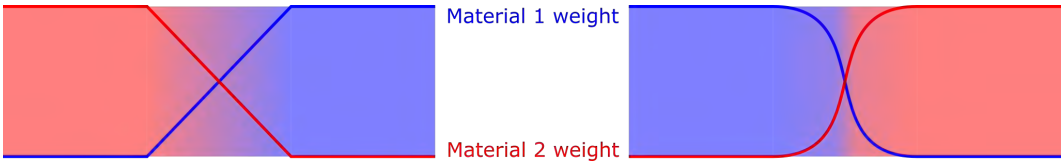


Figure 10.13: Blend diagram with linear interpolation (left) and sigmoid interpolation (right).

While being fairly easy to calculate, and do not require extra information, the results of this blending will be blurry. This problem can be solved by increasing the contrast between the material weights. However, this still results in unnatural changes of ground type. This blending will therefore be used from far away, when the details are not important.

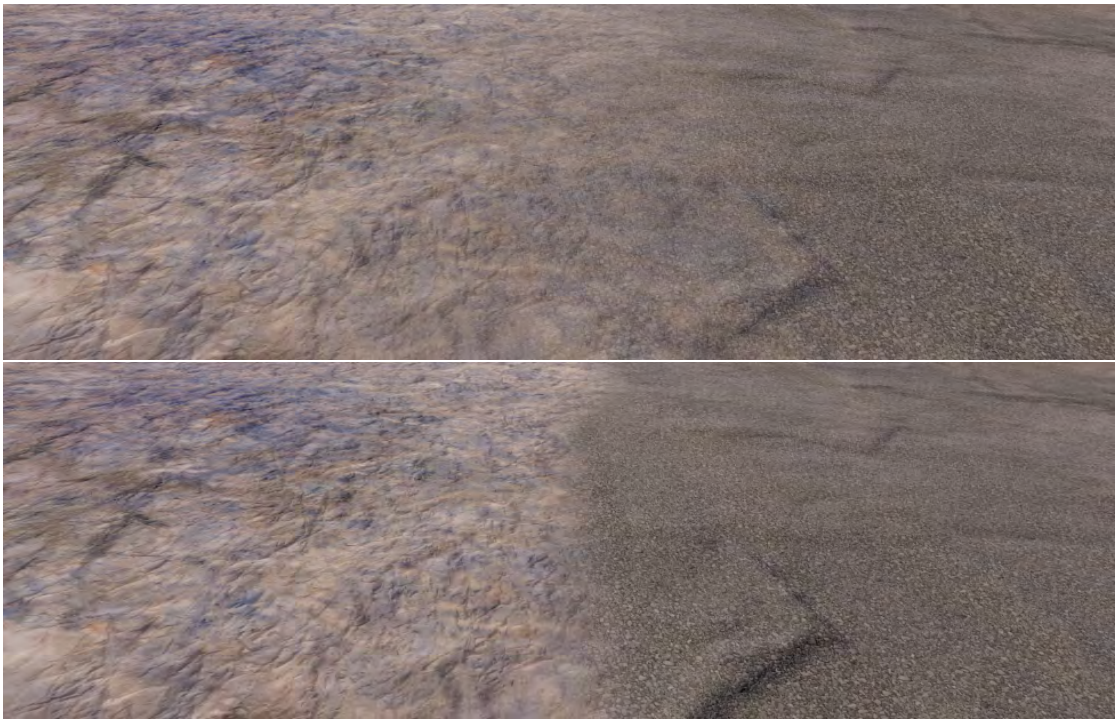


Figure 10.14: Interpolated blend between rock and grass with low weight contrast (top), and high weight contrast (bottom).

10.7.2 Height-Based Blending

Because the interpolated blend produces blur, it is not suitable for close-up views. This is why we are using a height-based blending for the more detailed views. Contrary to interpolated blend, this blending does not mix materials, but choose which one to use based on their weight. This weight is simply calculated as the multiplication of the weight of the interpolated blending, and the value from the material height texture. This allow us to have a more natural transition between different materials.

$$M = m_k \text{ if } \forall_{i \in N \setminus k} \; h_k w_k > h_i w_i$$

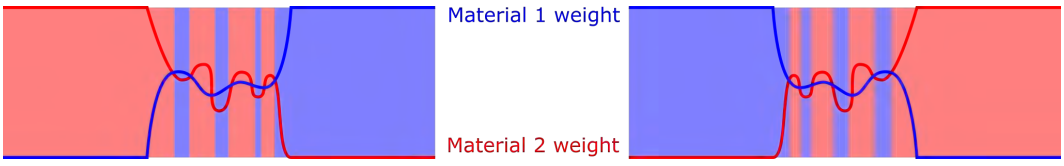


Figure 10.15: Height-based (left) and hybrid (right) blend diagram.

While this blending gives better results, it produces plenty of aliasing and flickering in the distance, as seen in Figure 10.16 top. Indeed, as this method choose one material or another, one pixel can drastically change color depending on which material it is. This problem can be solved by using a hybrid height/interpolated blending. This hybrid version chooses one material over

another if their weight difference is bigger than a certain threshold which is based on the distance. If the difference is below the threshold, the materials are mixed using a linear interpolation.

```
Material HybridBlend(Material Material0, Material Material1){
    float heightBlend = Depth / LODDistance2;

    if (Material1.w == 0 || Material0.w - Material1.w >= heightBlend)
        return Material0
    else if(Material0.w == 0 || Material1.w - Material0.w >= heightBlend)
        return Material1;
    else
    {
        float w = saturate(1 - abs(Material0.w - Material1.w) / heightBlend);
        w = 0.5 * w*w;
        return lerpMaterial(Material0, Material1, w);
    }
}
```

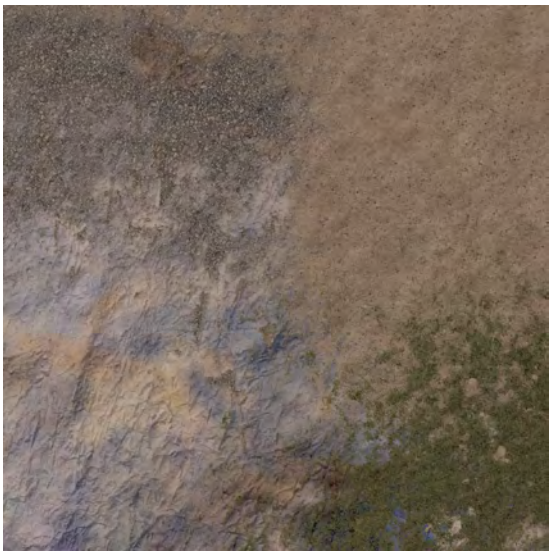
Listing 10.1: Hybrid Blend

Figure 10.16: Comparison between pure height-based blending (top) and hybrid height/interpolated blend (bottom).

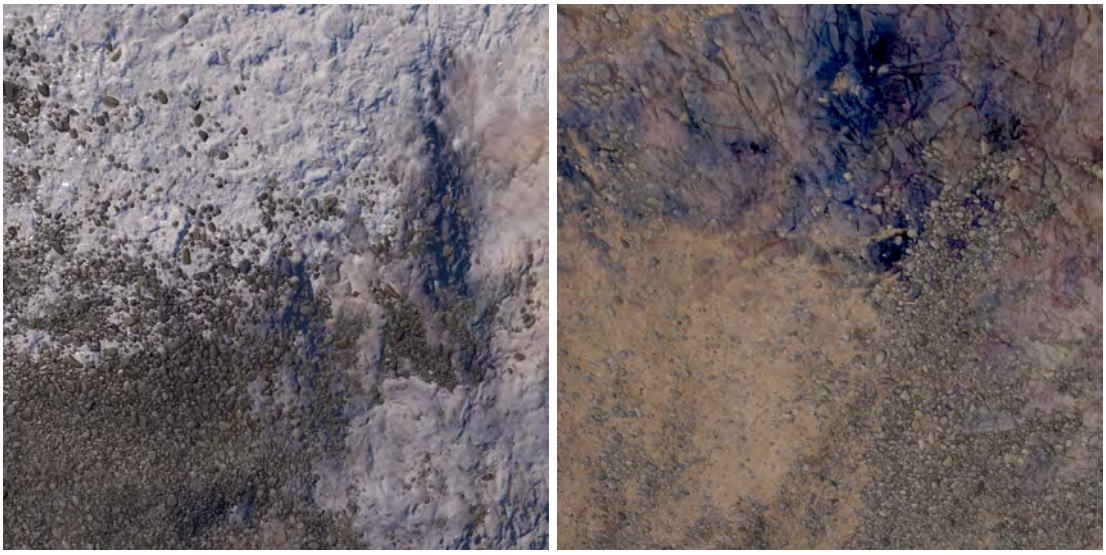
(a) Rock, grass, and gravel

(b) Soil, grass, and gravel

(c) Grass, and gravel



(d) Rock, grass, soil, and gravel



(e) Rock, Snow, and gravel (f) Rock, soil, and gravel

Figure 10.17: Examples of hybrid blending between diverse materials

10.8 Parallax

As we have access to the height information of the materials, we can use it to create a parallax effect. The parallax calculation is an iterative process that shift the uv coordinates used to sample a texture in order to fake a height displacement. This is useful to add extra bump details without creating more geometry.

While not being that visible on still images, parallax increase the realism and give a real sense of depth on the terrain. The parallax is only using the level 1 height. Indeed, the level 2 height have a too small effect to be visible.

```
1 for (int i = 0; i < PARALLAX_ITER; i++) {
2     float height = tex2D(_HeightTextures, UV*UVLargeMultiply).r;
3     float parralax = parralax_weight * (height - 0.5) * VertexNormal.z * View.xy;
4     UV -= parralax / (PARALLAX_ITER*UVLargeMultiply);
5 }
```

Listing 10.2: Parallax

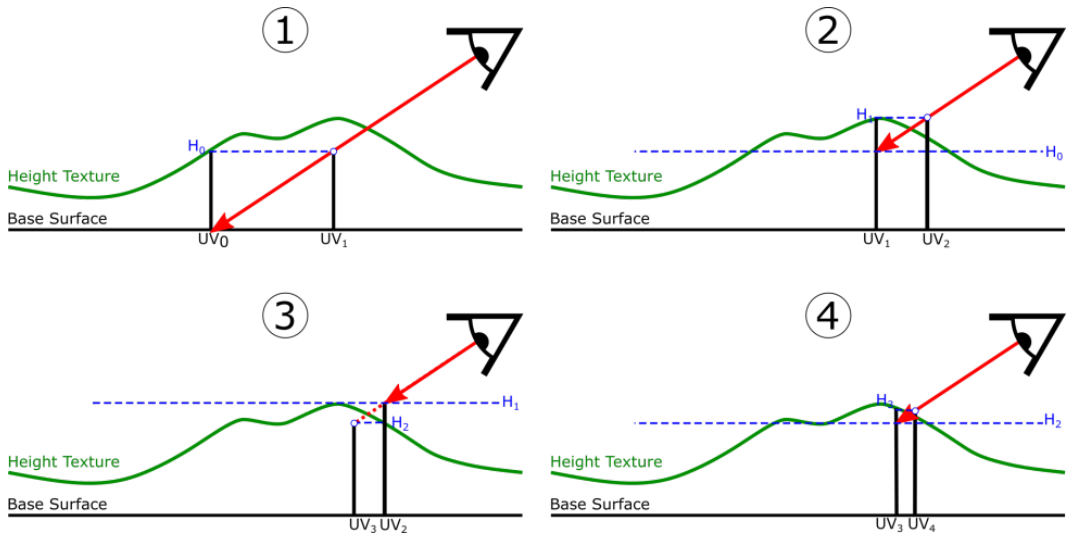


Figure 10.18: Iterative parallax process with 4 iterations.

Figure 10.19: Comparison of rock without (left) and with (right) parallax.

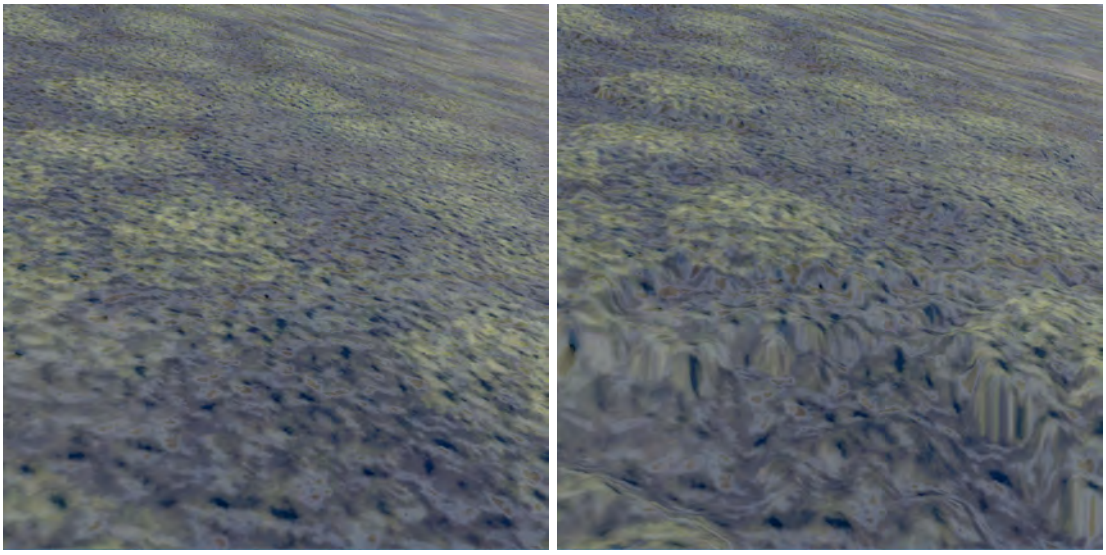


Figure 10.20: Comparison of forest without (left) and with (right) parallax.

Chapter 11

Water

11.1 Introduction

Contrary to ground materials, water is a dynamic element in constant movement and thus cannot be rendered realistically using simple textures. Therefore, we will use a procedural approach for this part. As seen in Darles et al. [2011], there exist multiple methods to simulate water. In our case, we only want to simulate and render the surface of the water. Therefore, we will use the rendering methods developed by Bruneton et al. [2010], which base its simulation upon Tessendorf et al. [2001].

11.2 Additional Variables

N	number total of waves
ω_i	frequency of wave i
h_i	amplitude of wave i
h_{max}	maximum amplitude allowed
N_{min} and N_{max}	minimum and maximum Nyquist value
λ_i	periodicity of waves i
λ_{min}, λ_{max}	minimum and maximum periodicity allowed
$g = 9.81$	gravity
t	time
W_{20}	wind speed at 20 meters above the surface
W_α	wind direction azimuth
W	wind rotation matrix
S	distance between two vertices of the terrain grid in meters

Table 11.1: Additional variables use for water rendering.

11.3 Wave Simulation

The water surface is modeled using n Gerstner waves. Gerstner wave, or trochoidal wave, describes the shape of the gravity waves [Gerstner, 1809]. It is defined in two dimensions as follows:

$$\mathbf{p}\begin{bmatrix} x \\ z \end{bmatrix} = \begin{bmatrix} x + h\sin(\omega t - kx) \\ h\cos(\omega t - kx) \end{bmatrix}$$

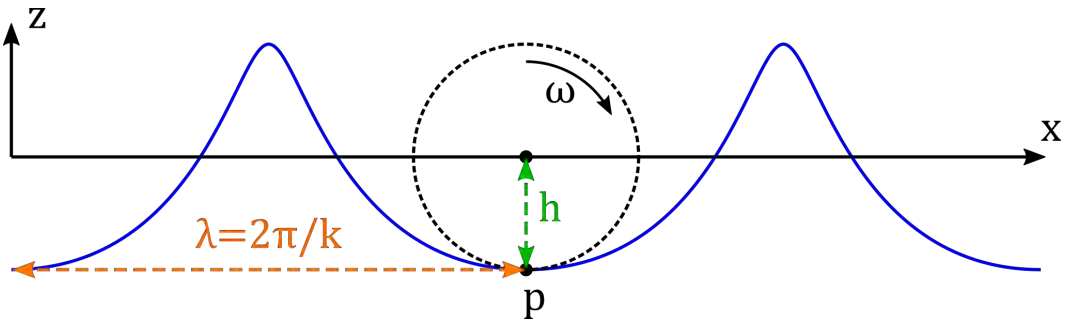


Figure 11.1: 2D Gerstner Wave.

Our water surface can then be simulated using several trochoidal waves with different amplitudes h_i and frequencies ω_i.

11.3.1 Frequencies

The frequencies are dependent on three user chosen parameters: the number of waves N, the minimum and maximum periodicity values, λ_{min} and λ_{max}. We can first calculate the periodicity λ_i of each wave as:

$$\lambda_i = 2^{1-\frac{i}{N}}m + \frac{i}{N}M, \text{ where } m = log_2\lambda_{min} \text{ and } M = log_2\lambda_{max}.$$

Then:

$$k_i = \frac{2\pi}{\lambda_i}$$

And finally:

$$\omega_i = \sqrt{gk_i}, \text{ with } g = 9.81$$

11.3.2 Amplitudes

The amplitudes depends on two user chosen parameters: the maximum amplitude h_{max}, and the wind speed at 20m above the water surface, V_{20}. We can first calculate the energy distribution of gravity waves by using the Pierson-Moskowitz formula [Pierson Jr and Moskowitz, 1964]:

$$H_i = \frac{\alpha g^2}{\omega_i^5}\, e^{\beta\left(\frac{\omega_0}{\omega_i}\right)^4}, \text{ where } \alpha = 8.1 \times 10^{-3} \text{ and } \beta = 0.74$$

And then we can calculate the amplitude h_i as:

$$h_i = 3\ h_{max}\ \sqrt{\frac{5s}{2}\sqrt{\frac{2\pi g}{\lambda_i}}\ H_i}, \text{ where } s = \frac{\lambda_{max} - \lambda_{min}}{N-1}$$

11.3.3 Waves Weight

As the same shader is used for both far and close views, the weight of each wave will depend on the view distance. Indeed, the furthest the viewer is, the less intense the waves with high frequencies need to be. This also means that we do not need to compute all waves at all distances. The weight and waves index boundary calculation will differ in the vertex and pixel shader.

Vertex Shader

In the vertex shader we will only select the frequencies high enough to actually displace the terrain grid. As the waves are sorted with the higher frequencies first, we only need to calculate the i_{min} value and we can set the i_{max} value to the number N of waves.

$$i_{min} = \left(log_2\left(N_{min}S\right) - m\right)\ \frac{N-1}{M-m}$$

$$i_{max} = N$$

The weight in the vertex shader is calculated as follow:

$$w_i = \mathbf{w}\left(N_{min}, N_{max}, \frac{2\pi g}{\omega_i^2 S}\right)$$

With

$$\mathbf{w}(a,b,x) = 3\bar{x}^2 - 2\bar{x}^3, \text{ and } \bar{x} = clamp\left(\frac{x-a}{b-a}, 0, 1\right)$$

Pixel Shader

As the lower frequencies waves are taken care of in the vertex shader, we only need to look at the higher frequencies waves in the pixel shader. However, we will still limit the frequencies chosen depending on the distance. Indeed, the highest frequencies waves can create aliasing in the distance.

$$i_{min} = \left(log_2\left(\frac{N_{min}S}{L}\right) - m\right)\ \frac{N-1}{M-m}$$

$$i_{max} = \left(log_2\left(N_{max}S\right) - m\right)\ \frac{N-1}{M-m}$$

With:

$$L = \sqrt{\tan^{-1}\left(\frac{S}{Z}\right)\ \frac{2C_h}{C_\alpha}\ \cos\theta_v}$$

The weight is decomposed into two components that are calculated in a similar fashion:

$$w_i = (1 - w_p)w_n$$

Where:

$$w_p = \mathbf{w}\left(N_{min}, N_{max}, \frac{2\pi g}{\omega_i^2 S}\right)$$

$$w_n = \mathbf{w}\left(N_{min}, N_{max}, \frac{2\pi g}{\omega_i^2 SL}\right)$$

11.3.4 Waves Dispersion

By simply using the Gerstner waves formula, we will obtain two dimensional waves. To have a three dimensions results, we can rotate each wave randomly around the z axis. To do that, we will calculate a rotation factor k_θ. This value depends on two user defined parameters: the waves dispersion D, and the number of different possible angles A. A value D of 0 will result in a perfectly uniform waves direction

$k_{\theta_i} = \frac{D\theta_i + \xi \frac{1.2}{A}}{1 + 40\left(\frac{\omega_0}{\omega_i}\right)^4}$, with $\xi \in [-1, 1]$ being a random number and $\theta_i \in [0, \pi]$ an angle

Using this orientation value, we can now calculate the final wave simulation. Firstly, we can calculate the phase φ_i at position (x, y) and time t of each wave as:

$$\varphi_i(x, y) = \omega_i t - \begin{pmatrix} k_i \cos k_\theta \\ k_i \sin k_\theta \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix}$$

Finally, the wave position can be expressed as:

$$\mathbf{p}\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} x + \sum_{i_{min}}^{i_{max}} w_i \; k_i h_i \; \frac{g}{\omega_i^2} \; \sin\left(\varphi_i(x, y)\right) \cos k_{\theta_i} \\ y + \sum_{i_{min}}^{i_{max}} w_i \; k_i h_i \; \frac{g}{\omega_i^2} \; \sin\left(\varphi_i(x, y)\right) \sin k_{\theta_i} \\ z + \sum_{i_{min}}^{i_{max}} w_i \; h_i \; \cos\left(\varphi_i(x, y)\right) \end{bmatrix}$$

In the same way we calculated the position, we can conveniently derive the calculation to get the normals of the waves that we will use when rendering.

$$\partial\mathbf{p}\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} -\sum_{i_{min}}^{i_{max}} w_i \; k_i h_i \; \frac{g}{\omega_i^2} \; \cos\left(\varphi_i(x, y)\right) \cos k_{\theta_i} \\ -\sum_{i_{min}}^{i_{max}} w_i \; k_i h_i \; \frac{g}{\omega_i^2} \; \cos\left(\varphi_i(x, y)\right) \sin k_{\theta_i} \\ \sum_{i_{min}}^{i_{max}} w_i \; h_i \; \sin\left(\varphi_i(x, y)\right) \end{bmatrix}$$

$$\frac{\partial \mathbf{p}}{\partial x} = k_i \cos k_{\theta_i} \; \partial P$$

$$\frac{\partial \mathbf{p}}{\partial y} = k_i \sin k_{\theta_i} \; \partial P$$

$$\mathbf{n} = \frac{\partial \mathbf{p}}{\partial x} \cdot \frac{\partial \mathbf{p}}{\partial y}$$

11.3.5 Wind Direction

We can control the average wave direction by specifying the wind rotation W_α. The rotation matrix associated to the wind can be calculated as:

$$W = \begin{pmatrix} \cos W_\alpha & \sin W_\alpha & 0 \\ -\sin W_\alpha & \cos W_\alpha & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Using the rotation matrix, we can transform the previously obtained normals for them to follow the wind direction:

$$n = Wn$$

11.4 Water BSDF

As water is a special kind of material, we cannot simply use the BRDF defined in Chapter 9. Instead we need to use a special BSDF that considers the light scattering inside the water as well as the surface interaction. This BSDF can be separated in three components: the specular reflection of the sun, $f_{sun}(l, v)$, the reflection of the environment, $f_{sky}(l, v)$, and the color and light scattering inside the water, $f_{sea}(l, v)$.

$$f_{water}(l, v) = f_{sun}(l, v) + f_{sky}(l, v) + f_{sea}(l, v)$$

Before looking at each specific component, we can first define a few values that will be useful. The central limit theorem tells us that the sum of many trochoids gives a surface whose slopes follow a Gaussian distribution. We can thus calculate the variance of this distribution as the sum of the variance of each wave slope distribution. This calculation is done in tangent space.

$$\begin{bmatrix} \sigma_x^2 \\ \sigma_y^2 \end{bmatrix} = \begin{bmatrix} \sum_i^N -(k_i \cos k_{\theta_i})^2 & \left(\sqrt{1 - w_k^2} - \sqrt{1 - k_h^2}\right) \\ \sum_i^N -(k_i \sin k_{\theta_i})^2 & \left(\sqrt{1 - w_k^2} - \sqrt{1 - k_h^2}\right) \end{bmatrix}$$

Where:

$$w_k = (1 - w_n)k_h$$

$$k_h = \frac{h_i \omega_i^2}{g}$$

Additionally, we can calculate the tangents of the slope:

$$T_y = N \times W[0]$$

$$T_x = T_y \times n$$

11.4.1 Sun Specular Reflection

As described in Bruneton et al. [2010], the BRDF use for the specular reflection was initially described by Ross et al. [2005]. This BRDF models accurately rough surfaces whose slopes follow a Gaussian distribution.

$$f_{sun} = \frac{F(\theta_d) e^{-0.5 \frac{\zeta_x^2/\sigma_x^2 + \zeta_y^2/\sigma_y^2}{2\pi\sigma_x\sigma_y}}}{4(1 + \Lambda(a_v) + \Lambda(a_l)\cos\theta_v \cos\theta_h{}^4}$$

Where:

$$\zeta_x = \frac{h \cdot T_x}{\cos\theta_h}$$

$$\zeta_y = \frac{h \cdot T_y}{\cos\theta_h}$$

And:

$$\Lambda(a_i) = \frac{e^{-a_i^2} - a_i\sqrt{\pi}\ \mathrm{erfc}(a_i)}{2a_i\sqrt{\pi}}, \text{ with } a_i = \frac{1}{\sqrt{2\tan\theta_i\ (\sigma_x^2\cos^2\Phi_i + \sigma_y^2\sin^2\Phi_i)}}$$

Φ_v and Φ_l being the angle between v, respectively l, and T_x.

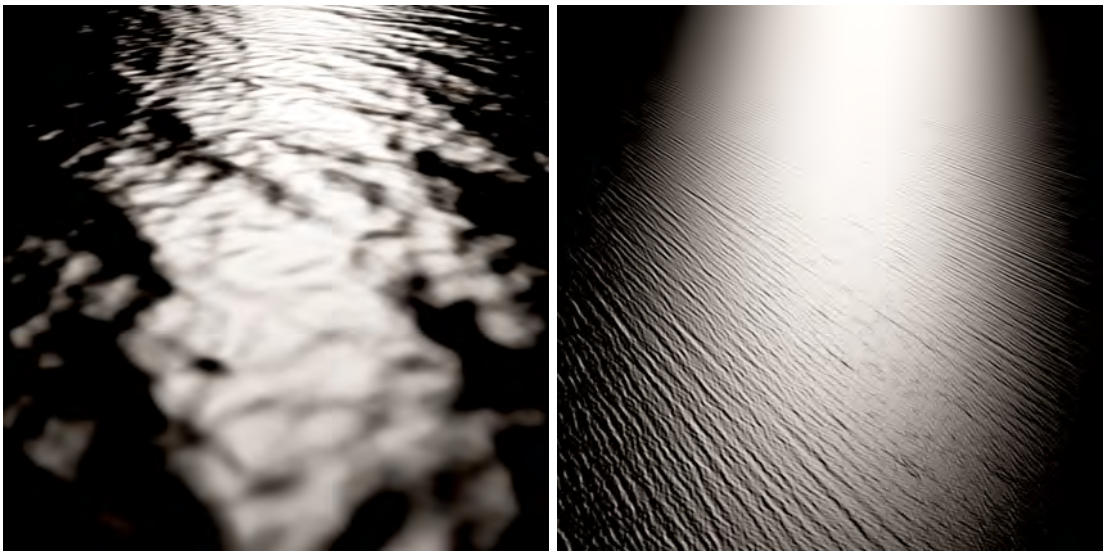


Figure 11.2: Sun specular reflection from close up (left) and far away (right) views.

11.4.2 Environment Reflection

To calculate the environment reflection, we can use a similar approach than in Section 9.4.2 by sampling a mipmap level of the environment cube map. The selected mipmap level can be calculated as:

$$MipMapLevel = max(\partial u_x/2.2, \partial u_y/2.2) * MipMapCount$$

Where:

$$\partial u_x = \frac{\sigma_x}{\epsilon}\left(U\left(\begin{bmatrix} \epsilon \\ 0 \end{bmatrix}\right) - U_0\right)$$

$$\partial u_y = \frac{\sigma_y}{\epsilon}\left(U\left(\begin{bmatrix} 0 \\ \epsilon \end{bmatrix}\right) - U_0\right)$$

With:

$$\epsilon = 0.001$$
$$U_0 = U\left(\begin{bmatrix} 0 \\ 0 \end{bmatrix}\right)$$

And:

$$U(\zeta) = \begin{bmatrix} R_x \\ R_y \end{bmatrix} / (1 + R_z)$$

$$R = 2(P \cdot v)P - v$$
$$P = p_x T_x + p_y T_y + p_z n$$

$$p = normalize\left(\begin{bmatrix} -\zeta_x \\ -\zeta_y \\ 0 \end{bmatrix}\right)$$

The resulting color will then be multiplied by the Fresnel factor. Contrary to the previous use of the Fresnel, we won't use the Schlick approximation for this part. Indeed, we will instead use the average of the Fresnel reflectance as described in Bruneton et al. [2010]:

$$\bar{F} = F_0 + (1 - F_0)\frac{(1 - \cos\theta_v)^{5e^{-2.69\sigma_v}}}{1 + 22.7\sigma_v^1.5}$$

Where σ_v^2 is the slope variance in the view direction which can be obtained as:

$$\sigma_v^2 = \frac{v_x^2\sigma_x^2}{1 - v_z^2} + \frac{v_y^2\sigma_y^2}{1 - v_z^2}$$

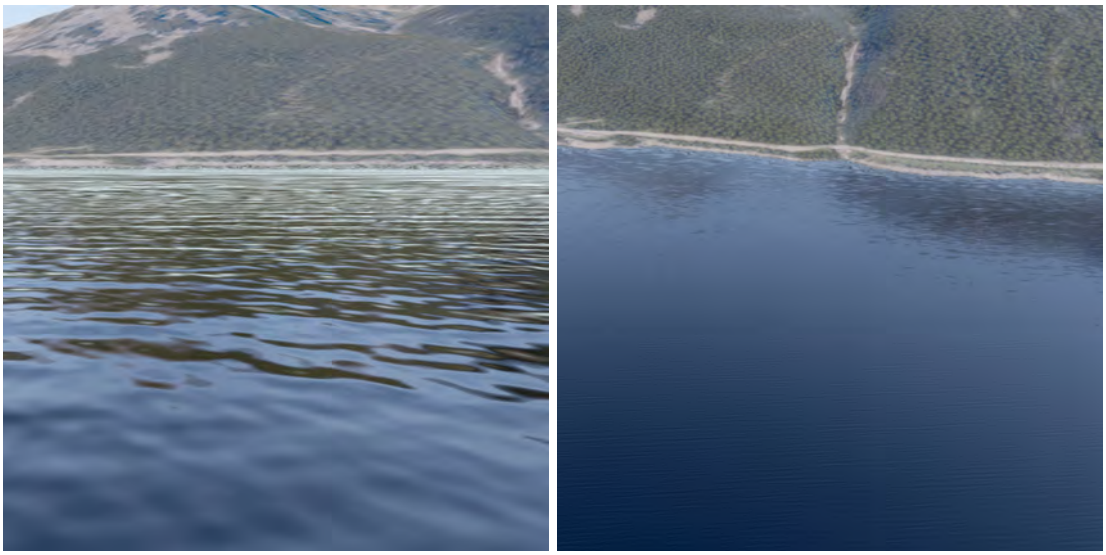


Figure 11.3: Environment reflection from close up (left) and far away (right) views.

11.4.3 Diffuse Color

The third and final component of the water BSDF is the "diffuse" color from the water. This part is essentially based on the scattering of the light inside the water. While complex calculations involving volume scattering and absorption would give physically accurate and realistic results, it would be impracticable in real-time. Luckily, we can fake the in-scattering with ease. Indeed, for now, we will consider the water to be infinitely deep and to be nothing to see through it. Therefore, the color is only influence by two factors: the color from the sky and the tint of the water.

The sky color comes from a simple gradient texture. The color depends on the sun zenith angle. Indeed, the higher the sun is the more the sky has light blue color. The tint of the water is a parameter defined by the user, it is usually a blue or green color depending on the type of water we want to render.

$$f_{sea}(l, v) = \frac{1}{\pi}(1 - F)\ sky(l_z)\ tint$$

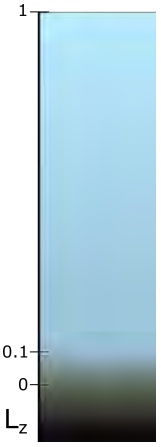


Figure 11.4: Sky color dependent of the sun zenith angle

11.4.4 Final BSDF

Now that all three parts are defined, we can add them together to have our final water BSDF.

Figure 11.5: Full water BSDF from close up (left) and far away (right) views.

11.5 Shores

The interaction between water and ground materials is special. Indeed, to be realistic, we have to consider that we can see the ground through the water near the shores. Therefore, we cannot simply use the same hybrid blending techniques as for the other materials. Instead we will develop another technique which is also based on the height of the materials. Indeed, as we have

access to the height h_w of the water from the simulation and the height h_m from the material on the shore, we can use them to create a realistic shore by having the ground material slowly fading under the water.

$$Color = \begin{cases} Color_m & \text{if } h_m \geq h_w \\ Color_w & \text{if } h_m + 1 \leq h_w \\ wColor_w + (1-w)Color_m & \text{else} \end{cases}$$

Where:

$$w = \text{clamp}\left((h_m - h_w)^2, 0, 1\right)$$

Figure 11.6: Ground material fading under the water based on height.

Additionally to the fade out, we can add a distortion effect to the ground below the surface of the water. Indeed, as the rays of light bend when entering/exiting the water surface, the ground will appear distorted. This can be done by shifting the UV coordinates for the material texture lookup.

$$uv = uv + d\ w\ n_{xy}$$

where n_{xy} are the x and y component of the water surface and d is a constant scaling value which is, in our case, set to $2 \cdot 10^{-4}$.

We can add a final effect to our shores. As water comes on the ground, it becomes wet, and thus darker and more specular. This effect appears only on the material that is not in the water. The wetness can be calculated as:

$$Wetness = \begin{cases} 1 - ((h_m - h_w)^2 - 1) & \text{if } h_m \geq h_w \\ 0 & \text{else} \end{cases}$$

Figure 11.7: Shore views from side (left) and top (right).

Chapter 12

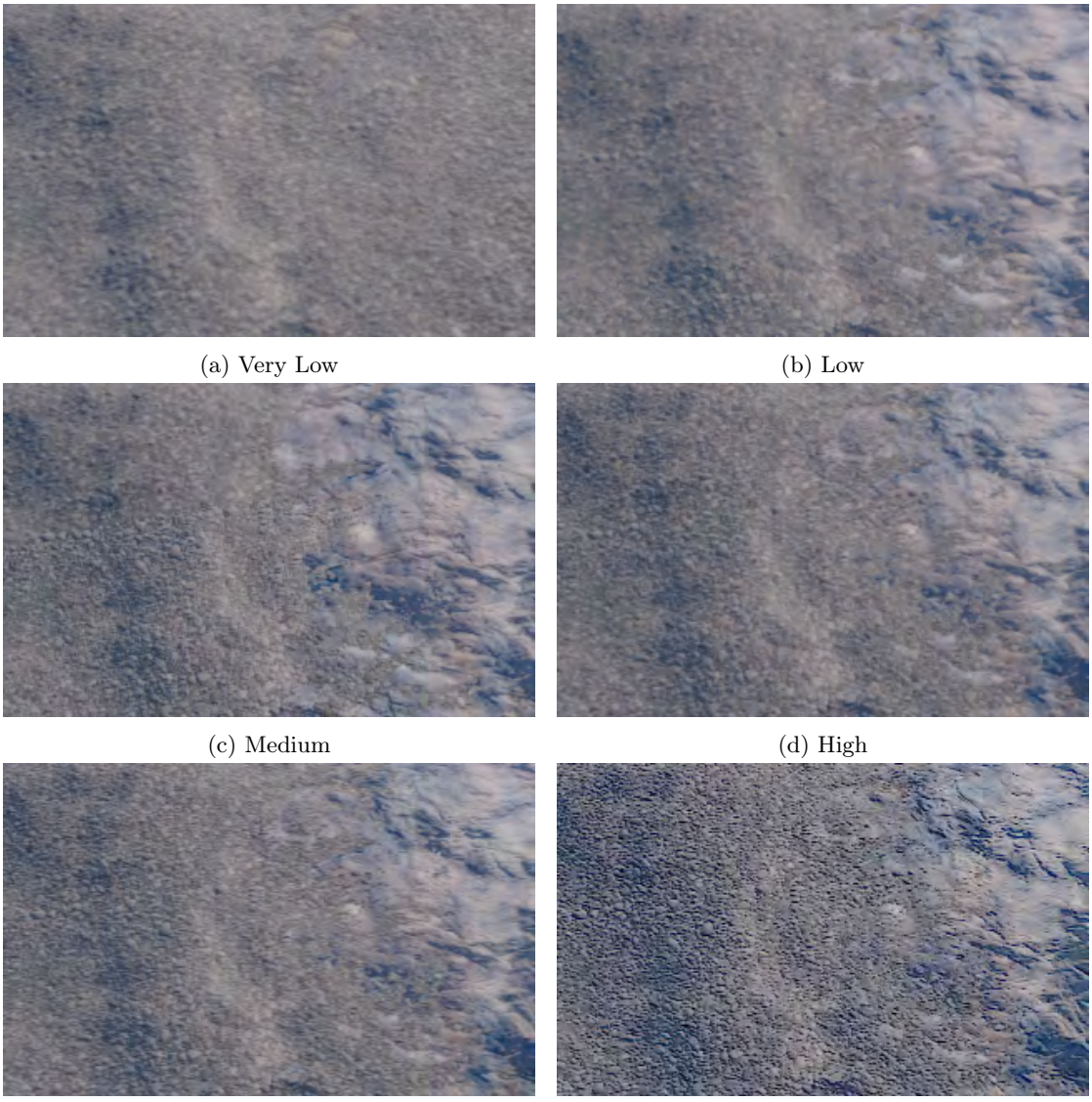
Performance

By adding PBR shading, procedural water, and high-quality textures, the overall rendering performance decreased. Indeed, while the satellite images could be rendered without a problem on most computer, rendering all the effects at their maximum capabilities required a recent dedicated graphic card. However, we can decrease some of the effects without losing to much quality. For this purpose, we created six levels of quality settings. The settings controlled by the quality levels are: the texture resolution (Tex Res), the global illumination diffuse ray count (GI Rays), the anti-aliasing (AA), and the shadows resolution (Sh Res) and cascade number (Sh Cas). A comparison of the different quality settings can be seen in Figure 12.1.

		Settings					
		Tex Res	GI Rays	AA	Shadow	Sh Res	Sh Cas
Quality	Very Low	128	4	FAAA[1]	No		
	Low	256	7	FAAA	No		
	Medium	512	14	SMAA[2]Low	Hard	Low	No
	High	1024	26	SMAA Med	Soft	Medium	2
	Very High	1024	43	SMAA High	Soft	High	4
	Ultra	1024	64	TAA[3]	Soft	Very High	4

Table 12.1: Quality settings

[1]Fast Approximate Anti-Aliasing
[2]Subpixel Morphological Anti-Aliasing
[3]Temporal Anti-Aliasing



(a) Very Low (b) Low

(c) Medium (d) High

(e) Very High (f) Ultra

Figure 12.1: Comparison of quality settings.

We have tested the performance on three different computers. The first one, a laptop, was equipped with an integrated Intel® HD Graphics 520 and a dedicated NVIDIA®GeForce® 940MX, the second one had an old NVIDIA® GeForce® GTX 580, while the third one was using a recent NVIDIA® GeForce® GTX 1080Ti. In the Table 12.2, you can see some performance results. The two first numbers are the time, in milliseconds, and the corresponding frames per second (fps) of the rendering of the camera view. The second number is the real fps of the whole scene. The difference between both is easily explained by the fact that the whole scene rendering need to render the cubemap, the shadow mask, and also include all the time spent on CPU tasks and data transfers.

As we can see, the goal of real-time rendering, at a minimum of 30 fps, is achieved with the

Quality	GPU: HD Graphics 520	GPU: 940MX	GPU: GTX 580	GPU: GTX 1080Ti
Very Low	13.1 (76) / 30	9.1 (110) / 43	4.1 (244) / 95	1.0 (991) / 319
Low	14.7 (68) / 22	10.1 (99) / 32	4.4 (230) / 74	1.35 (739) / 237
Medium	20.9 (48) / 13	12.8 (78) / 21	5.5 (182) / 49	1.9 (521))/ 140
High	62.4 (16) / 2	31.2 (32) / 4	6.7 (150) / 19	2.6 (378) / 110
Very High	142 (7) / 0	59.3 (17) / 1	8.5 (118) / 9	4.6 (219) / 79
Ultra	435 (2) / 0	163 (6) / 0	10.1 (99) / 4	5.2 (191) / 62

Table 12.2: Performance comparison between different GPUs and quality settings. First ans second numbers: time in ms spend on the GPU to render the camera point of view, with corresponding FPS in parentheses. Third number: true FPS.

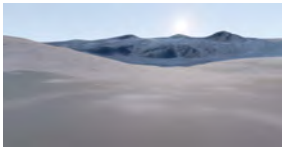
lower quality settings using a 2 years old laptop GPU, or a desktop GPU from 8 years ago. On the other hand, a recent state of the art GPU has no problem rendering at more than 60 fps, even with the best quality settings. However, an integrated graphic card such as the HD Graphics 520 barely achieve a 30 fps with the lowest setting.

Several others quality settings could be proposed to improve the performance. Indeed, reducing the cubemap size, which is by default 512x512 pixels for each side, could, for example, have a gain going from 2 (256x256) to 8 (64x64) fps. Moreover, we could replace the global illumination, especially the diffuse part which is expensive to compute, by a constant value. This would reduce slightly the realistic lighting but would have a fps gain from 3 to 5. Disabling the parallax could also help by adding 1 to 3 fps. By adding these small changes, the integrated GPU would be more viable and have more margin.

Chapter 13

Results

By putting all the pieces together, we can have some final render results. We can see in the figures below some comparisons "before/after" from which we can clearly see the improvements that our shader brings. While the changes in the distance are subtle, the changes in the foreground are well visible.



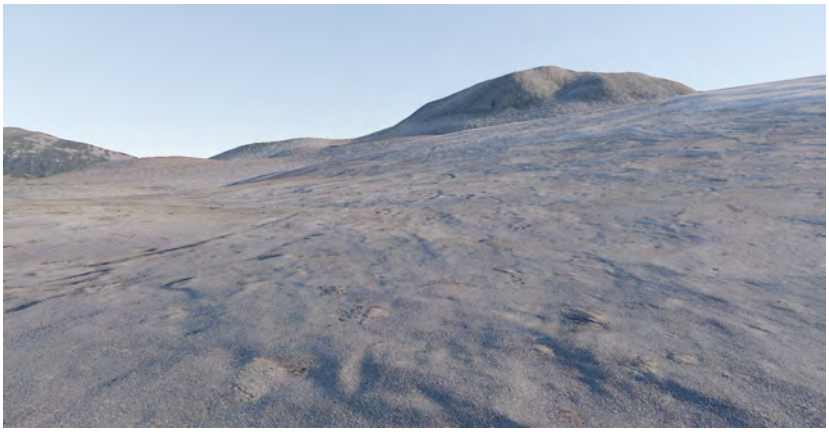


Figure 13.1: Comparison of renders with only the satellite image as material (left) versus the render with our materials (right).

Part III

Conclusion

Chapter 14

Summary

During this thesis, we have discussed a number of different techniques and algorithms to achieve our goal of realistically rendering landscapes using GIS data. In the first part, we looked at how we could infer ground material types from satellite imagery by using machine learning. After having described the two most used algorithm for land cover classification, we briefly looked at the features that could be extracted from the satellite images and height data. To construct our classifier, we compare the Decision Forest and Support Vector Machine and concluded that the former performed much better than the latter. Indeed, the Decision Forest achieves a 98.18% of accuracy while the SVM with Gaussian kernel only had an 82.36% of accuracy. We then looked at the contribution of each features to the accuracy and seen that some features were slightly reducing the global accuracy. In the end we were able to build a classifier based on the Decision Forest using 80 trees with an average accuracy of 98.76% and a Kappa value of 0.99.

In the second part, we tackled the problem of realistically rendering the landscape using the results from the previously found classifier. We first discussed the concept of Physically Based Rendering and how the Disney Principled BSDF could be used in real-time to achieve realistic shading. We also looked at how we could fake global illumination to have a more realistic lighting. Once our BRDF was defined, we presented our multi-scale material technique involving multiple layer of textures. We then described how we could blend materials with a realistic interaction by using the height of the materials. In the last chapter, we finally described a full procedural water simulation and rendering technique. In the end we tested the performance of the real-time rendering with a variety of GPUs.

Chapter 15

Conclusion and Future Work

We can conclude that our goal of improving the landscape rendering is partially achieved. The goal of real-time rendering is achieve and the enhancements are well visible, even in the lower quality settings. However, while being better than before, the final results are not realistic enough. Nevertheless, the water simulation and rendering is certainly the features that improve the most the results. The hybrid material blending is also giving a more realistic way to transition between terrain types. While these features brings a lot, there is still space for improvements. In the following sections, we will propose some topics that would be interesting to investigate to improve the landscape rendering.

15.1 Forest

The forests were given the same treatment as every other land cover in term of shading. However, they are much more complex than simple land cover. Indeed, the trees that compose a forest cannot be represented realistically by only using textures, at least not as it was done in this thesis. Several techniques to render realistic forest exist, however the problem of multi-scale is not widely research. Nevertheless, Bruneton and Neyret developed a technique which would be a very good fit for this goal [Bruneton and Neyret, 2012].

15.2 Populating Terrain

Aside from the trees, other element could add realism to the rendering. Indeed, by using only textures, our rendered ground lacks the details that 3D objects could bring. Several kinds of object can be added, such as bushes, rocks, flowers, and grass strands.

15.3 Correcting Water Height

As we can see in Figure 15.1, the lake seems to climb on the shore. This problem, probably due to the fact that the terrain mesh is low-poly and that the height data lacks in precision, could be solved by correcting the heightmap before rendering, or correcting the classification at render time. Indeed, we could for example use the classification result to set the same height value for every contiguous water pixels. On the other hand, we could correct the classification at render time by discarding water surface that are not flat.

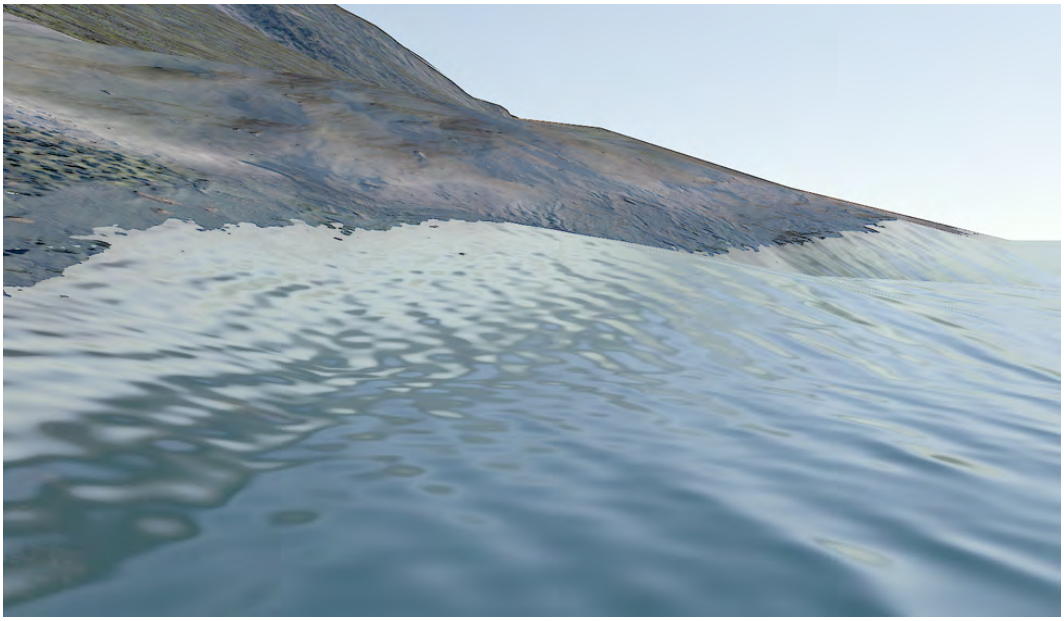


Figure 15.1: Ground material fading under the water based on height.

15.4 Atmospheric Effect

During the thesis we concentrate on the terrain. However, to have fully realistic landscape, it is also important to have atmospheric effect such as haze, fog, and clouds.

15.5 Generalization to other part of the world

During this thesis, we focus on mountain type landscapes, such as the Swiss Alps. To classify other part of the world, we would need to train new classifiers, and possibly adding new classes, such as sand, or even man-made structures. One approach to classify the entire planet would be to create one classifier that can be used anywhere in the world. Other features such as latitude and longitude could be added to encode different region of the world into the classifier. Another approach would be to create different classifiers for different landscape type: mountain, desert, tropical, etc. The choice of landscape type could be done manually, or with a machine learning algorithm.

15.6 Satellite Images Issues

Because the satellite images are taken from space, there qualities is subject to the weather in the region being photographed. If it is cloudy, the images will not be able to see the ground, and will only take pictures of the cloud. Another problem is that the images do not come from a single satellite, but from several different one. This can create problem such as abrupt color change. A third problem is that satellite take pictures all year long. This can create sudden seasons change in the imagery. These issues could be solve by better selection of the images, and or some processing before usage.

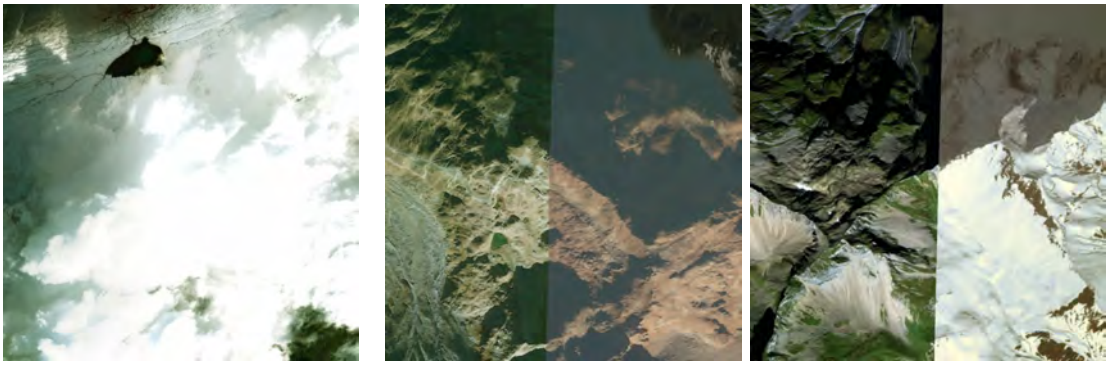


Figure 15.2: Three problems with satellite images: cloud (left), color changes (middle), and season changes (right).

15.7 Moving to WebGL

While the initial goal was to do the rendering directly on the web, we chose to prototype it using Unity. The porting to WebGL should not be too hard. Indeed, while the shader was written in HLSL, nothing specific to DirectX and only the most basic integrated functions were used. Some of the functions and parameters were given by Unity, but there should not be a problem to reproduce them for the web. As it was done in Unity, all the textures should be packed in texture arrays, as the maximum number guaranteed of textures supported by shader is 8, and we have more than 30 textures in total. On of the challenge that would come when moving to the web is the data transfer. The high number of high definition textures can be a problem. This can be partially solve by compressing the textures, and use lower resolution.

List of Figures

List of Tables

Bibliography

ALGLIB. Alglib documentation, March 2018. URL `http://www.alglib.net/docs.php`.

Oscar Argudo, Marc Comino, Antonio Chica, Carlos Andújar, and Felipe Lumbreras. Segmentation of aerial images for plausible detail synthesis. *Computers & Graphics*, 2017.

Simon Bernard, Sébastien Adam, and Laurent Heutte. Dynamic random forests. *Pattern Recognition Letters*, 33(12):1580–1586, 2012.

Eric Bruneton and Fabrice Neyret. Real-time realistic rendering and lighting of forests. In *Computer Graphics Forum*, volume 31, pages 373–382. Wiley Online Library, 2012.

Eric Bruneton, Fabrice Neyret, and Nicolas Holzschuch. Real-time realistic ocean lighting using seamless transitions from geometry to brdf. In *Computer Graphics Forum*, volume 29, pages 487–496. Wiley Online Library, 2010.

Brent Burley and Walt Disney Animation Studios. Physically-based shading at disney. In *ACM SIGGRAPH*, volume 2012, pages 1–7, 2012.

Jacob Cohen. A coefficient of agreement for nominal scales. *Educational and psychological measurement*, 20(1):37–46, 1960.

Russell G Congalton. A review of assessing the accuracy of classifications of remotely sensed data. *Remote sensing of environment*, 37(1):35–46, 1991.

Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine learning*, 20(3):273–297, 1995.

Emmanuelle Darles, Benoît Crespin, Djamchid Ghazanfarpour, and Jean-Christophe Gonzato. A survey of ocean simulation and rendering techniques in computer graphics. In *Computer Graphics Forum*, volume 30, pages 43–60. Wiley Online Library, 2011.

Echoview. Glcm texture feature, March 2018. URL `http://support.echoview.com/WebHelp/Windows_and_Dialog_Boxes/Dialog_Boxes/Variable_properties_dialog_box/Operator_pages/GLCM_Texture_Features.htm`.

Accord.NET Framework. Accord.net documentation, March 2018. URL `http://accord-framework.net/docs/html/R_Project_Accord_NET.htm`.

Franz Gerstner. Theorie der wellen. *Annalen der Physik*, 32(8):412–445, 1809.

Bruce Hapke. *Theory of reflectance and emittance spectroscopy*. Cambridge university press, 2012.

Robert M Haralick, K Shanmugam, Its'Hak Dinstein, et al. Textural features for image classification. *IEEE Transactions on systems, man, and cybernetics*, 3(6):610–621, 1973.

Tin Kam Ho. Random decision forests. In *Document analysis and recognition, 1995., proceedings of the third international conference on*, volume 1, pages 278–282. IEEE, 1995.

C Huang, LS Davis, and JRG Townshend. An assessment of support vector machines for land cover classification. *International Journal of remote sensing*, 23(4):725–749, 2002.

S Lagarde and CD Rousiers. Moving frostbite to physically based rendering. *part of ACM SIGGRAPH2014 Course: Physically Based Shading in Theory and Practice*, 2014.

Big Vision LLC. Histogram of oriented gradients, March 2018. URL `https://www.learnopencv.com/histogram-of-oriented-gradients/`.

P Mohanaiah, P Sathyanarayana, and L GuruKumar. Image texture feature extraction using glcm approach. *International Journal of Scientific and Research Publications*, 3(5):1, 2013.

Benjamin Neukom. Gis-based landscape visualizations. Master's thesis, University of Applied Sciences and Arts, 2017.

John Richard Otukei and Thomas Blaschke. Land cover change assessment using decision trees, support vector machines and maximum likelihood classification algorithms. *International Journal of Applied Earth Observation and Geoinformation*, 12:S27–S31, 2010.

Willard J Pierson Jr and Lionel Moskowitz. A proposed spectral form for fully developed wind seas based on the similarity theory of sa kitaigorodskii. *Journal of geophysical research*, 69 (24):5181–5190, 1964.

Jure Ratković. *Physically based rendering.* PhD thesis, Fakultet elektrotehnike i računarstva, Sveučilište u Zagrebu, 2017.

Lior Rokach. Decision forest: Twenty years of research. *Information Fusion*, 27:111–125, 2016.

Vincent Ross, Denis Dion, and Guy Potvin. Detailed analytical approach to the gaussian surface bidirectional reflectance distribution function specular component applied to the sea surface. *JOSA A*, 22(11):2442–2453, 2005.

Christophe Schlick. An inexpensive brdf model for physically-based rendering. In *Computer graphics forum*, volume 13, pages 233–246. Wiley Online Library, 1994.

Jerry Tessendorf et al. Simulating ocean water. *Simulating nature: realistic and interactive techniques. SIGGRAPH*, 1(2):5, 2001.

Bruce Walter, Stephen R Marschner, Hongsong Li, and Kenneth E Torrance. Microfacet models for refraction through rough surfaces. In *Proceedings of the 18th Eurographics conference on Rendering Techniques*, pages 195–206. Eurographics Association, 2007.

Quentin Kuenlin

Marly, August 17th 2018

Appendix A

BRDF

```
float3 BRDF(Material m) {
    float3 N;
        N.x = dot(tspace0, m.Normal.xyz);
        N.y = dot(tspace1, m.Normal.xyz);
        N.z = dot(tspace2, m.Normal.xyz);

        float NdotV = abs(dot(N, V)) + 1e-5f;

        /* DIRECT SUN ILLUMINATION */
        float3 Direct = 0;
        if(Shadows > 0)
        {
                float LdotH = saturate(dot(L, H));
                float NdotH = saturate(dot(N, H));
                float NdotL = saturate(dot(N, L));

                float3 Fd = 0;
                float3 Fr = 0;
                if (NdotL > 0 && NdotV > 0) {
                        //Disney Principled Diffuse
                        Fd = m.Albedo.rgb * Principled_DisneyDiffuse(NdotV, NdotL,
                             LdotH, m.Roughness);

                        //GGX Specular
                        float3 F = F_Schlick(LdotH, 0.04);
                        float G = V_SmithGGXCorrelated(NdotV, NdotL, 0.5 + m.
                            Roughness / 2.0);
                        float D = D_GGX(NdotH, m.Roughness);
                        Fr = D * F * G * M_1_PI * NdotL;
                }

                Direct = float4(_LightColor0 * (Fd + Fr) * Shadows, 1.0);
        }

        /* INDIRECT DIFFUSE */
        float3 tangentX, tangentY;
        GetLocalFrame(N, tangentX, tangentY);

        float2 randNum = InitRandom(N.xz * 0.5f + 0.5f);

        float r = 1.0f * _LightSampleCount / 64.0f;
        r = saturate(1 - r);
        r = r * r * r * r;
```

```
42
43          float3 indirectDiffuse = 0;
44          [loop]
45          for (int i = 0; i < _LightSampleCount; i++) {
46                  float2 u = Hammersley2d(i, _LightSampleCount);
47                  u = frac(u + randNum + 0.5f);
48                  float3 L;
49                  float NdotL;
50                  float weightOverPdf;
51
52                  // for Disney we still use a Cosine importance sampling, true
                        Disney importance sampling imply a look up table
53                  ImportanceSampleLambert(u, N, tangentX, tangentY, L, NdotL,
                        weightOverPdf);
54
55                  NdotL = saturate(dot(N, L));
56                  if (NdotL > 0 && NdotV > 0) {
57                          float3 H = normalize(V + L);
58                          float LdotH = saturate(dot(L, H));
59                          float3 NdotH = saturate(dot(N, H));
60
61                          float3 sky = SampleReflection(L, r);
62                          indirectDiffuse += sky * m.Albedo.rgb *
                                Principled_DisneyDiffuse(NdotV, NdotL, LdotH, m.
                                Roughness) * weightOverPdf;
63                  }
64          }
65
66          /* INDIRECT SPECULAR */
67          float3 indirectSpecular = 0;
68          float3 L = reflect(-V, N);
69          float NdotL = saturate(dot(N, L));
70          if (NdotL > 0.0f) {
71                  float3 H = normalize(V + L);
72                  float LdotH = saturate(dot(L, H));
73                  float NdotH = saturate(dot(N, H));
74                  float weightOverPdf = 0;
75                  float VdotH = dot(V, H);
76
77                  float Vis = V_SmithGGXCorrelated(NdotL, NdotV, 0.5 + m.Roughness /
                        2.0);
78                  weightOverPdf = 4.0f * Vis * NdotL * VdotH / NdotH;
79
80                  if (weightOverPdf > 0)
81                  {
82                          float3 sky = SampleReflection(L.xyz, m.Roughness);
83                          indirectSpecular = F_Schlick(VdotH, 0.02) * sky *
                                weightOverPdf;
84                  }
85          }
86
87          return Direct + (Fd/_LightSampleCount + Fr);
88  }
```

Listing A.1: BRDF

Appendix B

Material Template

```
Material MaterialTemplate(float wetness, float height) {
    float2 UVLarge    = UV*UVLargeMultiply;

    float4 albedoL    = tex2D(ColorLarge, UVLarge);
    float4 albedoLM   = tex2DLod(ColorLarge, UVLarge, 9);
    float  Roughness  = tex2D(RoughnessLarge, UVLarge).x;
    float3 NormalL    = tex2D(NormalLarge, UVLarge);

    float4 Albedo     = ColorTransfer(satellite, albedoL, albedoLM);
    float  Roughness  = 0;
    float3 Normal     = float3(0, 1, 0);

    if (Depth < LODDistance1)
    {
        float2 UVDetails  = UV*UVDetailMultiply;

        float4 albedoD    = tex2Dlod(ColorDetails, UVDetails, TexLod);
        float4 albedoDM   = tex2Dlod(ColorDetails, UVDetails, TexLod+8;
        float  roughnessD = tex2Dlod(RoughnessDetails, UVDetails, TexLod).x;
        float3 normalD    = tex2DLod(NormalDetails, UVDetails, TexLod);

        float4 colorD     = ColorTransfer(albedo, albedoD, albedoDM);

        if (Depth >= LODDistance1 * (1-blendSize))
        {
            float w   = (Depth - LODDistance1*(1-blendSize));
                  W  /= (LODDistance1*blendSize);

            Albedo    = lerp(colorD, Albedo, w);
            Normal    = blendNormal(NormalL, NormalD, 1 - w);
            Roughness = lerp((roughnessL + roughnessD)/2, roughnessL, w);
        }
        else
        {
            Normal    = blendNormal(NormalL, NormalD, 1);
            Roughness = (roughnessL + roughnessD)/2;

            if (Depth < LODDistance0)
                Albedo = lerp(albedoD, colorD, (Depth) / (LODDistance0));
            else
                Albedo = colorD;
        }
    }
```

```
44
45      /* Wet parts are less rough and darker */
46      float wet = saturate(2.0*wetness - pow(1.2*(height), 2));
47      Roughness = lerp(Roughness, RoughnessModifier, RoughnessModifierStrength);
48      Roughness = lerp(Roughness, 0.1, wet);
49      Albedo    *= lerp(1.0, 0.6, wet);
50
51      /* Final Material */
52      Material m;
53      m.Albedo    = Albedo;
54      m.Normal    = Normal;
55      m.Roughness = Roughness;
56      return m;
57  }
```

Listing B.1: Material Template

Appendix C

Water Simulation and Rendering

```
1  void generateWaves()
2  {
3      float min = log(lambdaMin) / log(2.0f);
4      float max = log(lambdaMax) / log(2.0f);
5      float omega0 = 9.81f / v20;
6
7      float dangle = (1.5f / (float)(nbAngles / 2));
8      int[] index = new int[5]; // to hash angle order
9
10     for (int i = 0; i < nbAngles; i++)
11         index[i] = i;
12
13     for (int i = 0; i < nbWaves; ++i)
14     {
15         Vector4 waves = new Vector4();
16         float x = i / (nbWaves - 1.0f);
17
18         float lambda = pow(2.0f, (1.0f - x) * min + x * max);
19         float ktheta = random(0.0f, 1.0f) * waveDispersion;
20         float knorm = 2.0f * Mathf.PI / lambda;
21         float omega = sqrt(9.81f * knorm);
22         float amplitude_;
23
24         float step = (max - min) / (nbWaves - 1);
25
26         if ((i % (nbAngles)) == 0)
27         {
28             // scramble angle ordre
29             for (int k = 0; k < nbAngles; k++)
30             {   // do N swap in indices
31                 int n1 = (int)lrandom() % nbAngles, n2 = (int)lrandom() % nbAngles
                       , n;
32                 n = index[n1];
33                 index[n1] = index[n2];
34                 index[n2] = n;
35             }
36         }
37
38         ktheta = (angle(index[(i) % nbAngles], nbAngles) + 0.4f * srnd() * dangle)
               ;
39         ktheta *= waveDispersion;
40         ktheta /= (1.0f + 40.0f * pow(omega0 / omega, 4));
41
```

```
42          amplitude_ = (8.1e-3f * 9.81f * 9.81f) / pow(omega, 5);
43          amplitude_ *= exp(-0.74f * pow(omega0 / omega, 4));
44          amplitude_ *= 0.5f * sqrt(2 * Mathf.PI * 9.81f / lambda) * nbAngles * step
                ;
45          amplitude_ = 3 * amplitude * sqrt(amplitude_);
46
47          if (amplitude_ > 1.0f / knorm)
48              amplitude_ = 1.0f / knorm;
49          else if (amplitude_ < -1.0f / knorm)
50              amplitude_ = -1.0f / knorm;
51
52          waves.x = amplitude_;
53          waves.y = omega;
54          waves.z = knorm * Mathf.Cos(ktheta);
55          waves.w = knorm * Mathf.Sin(ktheta);
56
57          float tmp = (1.0f - sqrt(1.0f - knorm * knorm * amplitude_ * amplitude_));
58
59          wavesTex.SetPixel(i, 0, new Color(waves.x, waves.y, waves.z, waves.w));
60      }
61  }
```

Listing C.1: Wave Generation on the CPU

```
1   float4 Water(v2f input, float weight) {
2
3       //Get Values from Vertex Shader
4       float3 dPdu = input.dPdu;
5       float3 dPdv = input.dPdv;
6       float2 sigmaSq = input.sigmaSq;
7       float h = input.dP;
8
9       float iMAX = min(nbWaves - 1.0, ceil((log2(nyquistMax * lods.y) - lods.z) *
            lods.w));
10      float iMax = floor((log2(nyquistMin * Depth / input.lod) - lods.z) * lods.w);
11      float iMin = max(0.0, floor((log2(nyquistMin * lods.y / input.lod) - lods.z) *
             lods.w));
12
13      [loop]
14      for (float i = iMin; i <= iMAX; i += 1.0) {
15          float4 wt = tex2Dlod(wavesTex, float4((i + 0.5f) / nbWaves, 0, 0, 0));
16          float phase = wt.y * Time - dot(wt.zw, input.u);
17          float s = sin(phase);
18          float c = cos(phase);
19          float overk = 9.81 / (wt.y * wt.y);
20
21          float wp = smoothstep(nyquistMin, nyquistMax, (M_2PI) * overk / lods.y);
22          float wn = smoothstep(nyquistMin, nyquistMax, (M_2PI) * overk / lods.y *
                input.lod);
23
24          float3 factor = (1.0 - wp) * wn * wt.x * float3(wt.zw * overk, 1.0);
25
26          h += factor * float3(s, c, s);
27
28          float3 dPd = factor * float3(c, c, -s);
29          dPdu -= dPd * wt.z;
30          dPdv -= dPd * wt.w;
31
32          wt.zw *= overk;
33          float kh = i < iMax ? wt.x / overk : 0.0;
34          float wkh = (1.0 - wn) * kh;
```

```
35          sigmaSq -= float2(wt.z * wt.z, wt.w * wt.w) * (sqrt(1.0 - wkh * wkh) -
                sqrt(1.0 - kh * kh));
36      }
37
38      sigmaSq = max(sigmaSq, 2e-5);
39
40      //Normal in Wind space
41      float3 windNormal = normalize(cross(dPdu, dPdv));
42
43      //Normal in world space
44      float3 N = float3(mul(windToWorld, float4(windNormal.xy, 0, 0)).xy, windNormal
            .z);
45
46      //Tangents
47      float3 Ty = normalize(cross(N, float3(windToWorld[0].xy, 0.0)));
48      float3 Tx = cross(Ty, N);
49
50      float fresnel = 0.02 + 0.98 * meanFresnel(V, N, sigmaSq);
51
52      //Sun Specular Reflection
53      float3 Sun = SunSpecular(L, V, N, Tx, Ty, sigmaSq);
54
55      //Environment Reflection
56      float3 Env = fresnel * EnvReflection(V, N, Tx, Ty, sigmaSq);
57
58      //Water Diffuse
59      float3 Sea = (1 - fresnel) * WaterColor * Sky(L.y) * M_1_PI;
60
61      float4 WaterColor = float4(Lsea + Lsky + Lsun, 1.0);
62
63      return WaterColor;
64  }
65
66  float3 EnvReflection(float3 V, float3 N, float3 Tx, float3 Ty, float2 sigmaSq) {
67      const float eps = 0.001;
68      float3 wo = reflect(-V, N);
69
70      float2 u0 = U(float2(0,0), V, N, Tx, Ty);
71      float3 dux = float3(2.0 * (U(float2(eps, 0.0), V, N, Tx, Ty) - u0) / eps *
            sqrt(sigmaSq.x),0);
72      float3 duy = float3(2.0 * (U(float2(0.0, eps), V, N, Tx, Ty) - u0) / eps *
            sqrt(sigmaSq.y),0);
73      half mip = max(0.0, max(length(dux.xzy * (0.5 / 1.1)), length(duy.xzy * (0.5 /
             1.1)))*9);
74      float4 result = float4(UNITY_SAMPLE_TEXCUBE_LOD(unity_SpecCube0, wo.xzy, mip),
             1.0);
75      return result.rgb;
76  }
77
78  float SunSpecular(float3 L, float3 V, float3 N, float3 Tx, float3 Ty, float2
        sigmaSq) {
79      float3 H = normalize(L + V);
80      float zH = dot(H, N); // cos of facet normal zenith angle
81
82      float zetax = dot(H, Tx) / zH;
83      float zetay = dot(H, Ty) / zH;
84
85      float zL = dot(L, N); // cos of source zenith angle
86      float zV = dot(V, N); // cos of receiver zenith angle
87      float zH2 = zH * zH;
88
89      float p = exp(-0.5 * (zetax * zetax / sigmaSq.x + zetay * zetay / sigmaSq.y))
```

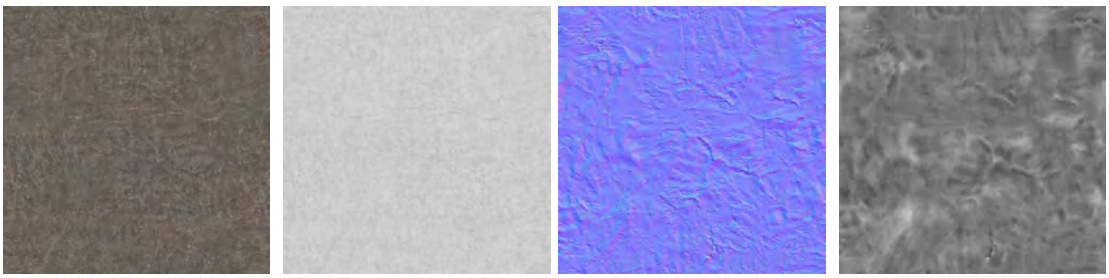
```
        / (M_2PI * sqrt(sigmaSq.x * sigmaSq.y));

    float tanV = atan2(dot(V, Ty), dot(V, Tx));
    float cosV2 = 1.0 / (1.0 + tanV * tanV);
    float sigmaV2 = sigmaSq.x * cosV2 + sigmaSq.y * (1.0 - cosV2);

    float tanL = atan2(dot(L, Ty), dot(L, Tx));
    float cosL2 = 1.0 / (1.0 + tanL * tanL);
    float sigmaL2 = sigmaSq.x * cosL2 + sigmaSq.y * (1.0 - cosL2);

    float fresnel = 0.02 + 0.98 * pow(1.0 - dot(V, H), 5.0);

    zL = max(zL, 0.01);
    zV = max(zV, 0.01);

    return fresnel * p / ((1.0 + Lambda(zL, sigmaL2) + Lambda(zV, sigmaV2)) * zV *
        zH2 * zH2 * 4.0);
}
```

Listing C.2: Wave Pixel Shader Simulation and Rendering

Appendix D

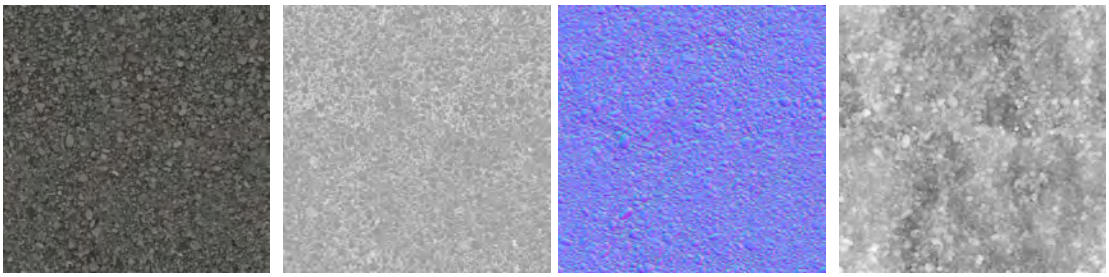
Textures

D.1 Gravel and Soil



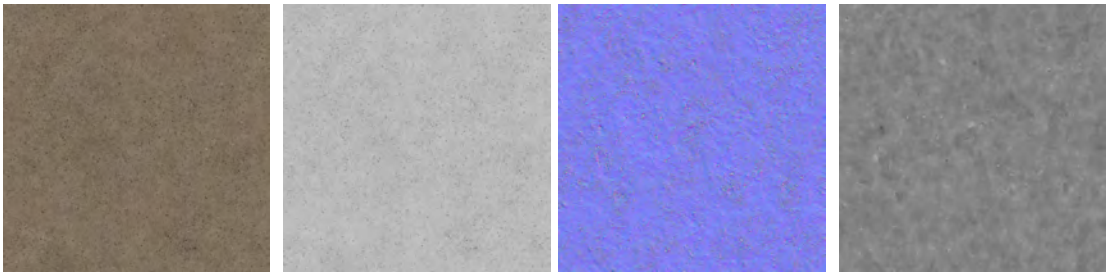
(a) Base color (b) Roughness (c) Normal (d) Height

Figure D.1: Level 1 gravel and soil textures.



(a) Base color (b) Roughness (c) Normal (d) Height

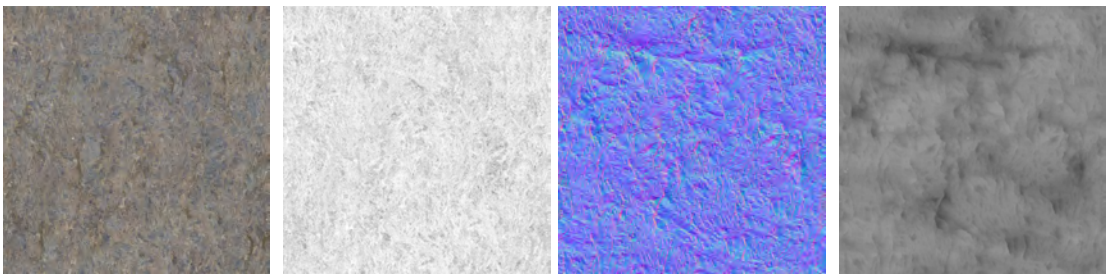
Figure D.2: Level 2 gravel textures.



(a) Base color (b) Roughness (c) Normal (d) Height

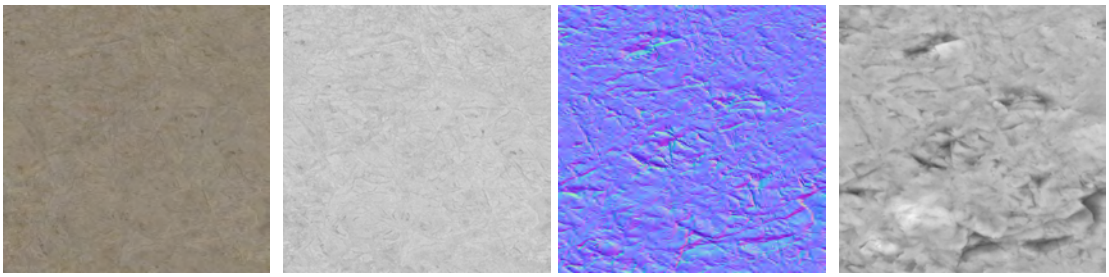
Figure D.3: Level 2 soil textures.

D.2 Rock



(a) Base color (b) Roughness (c) Normal (d) Height

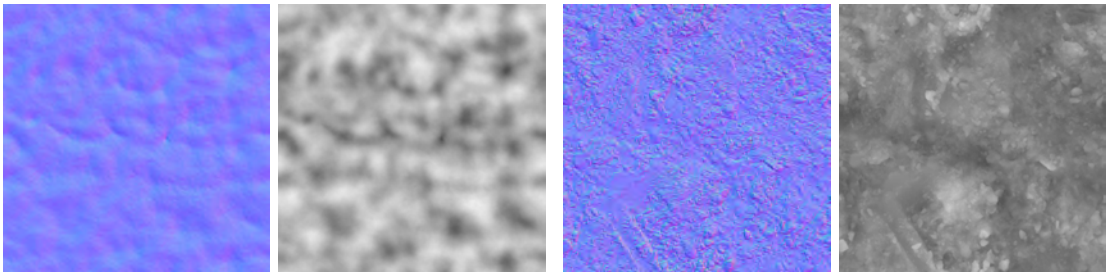
Figure D.4: Level 1 rock textures.



(a) Base color (b) Roughness (c) Normal (d) Height

Figure D.5: Level 2 rock textures.

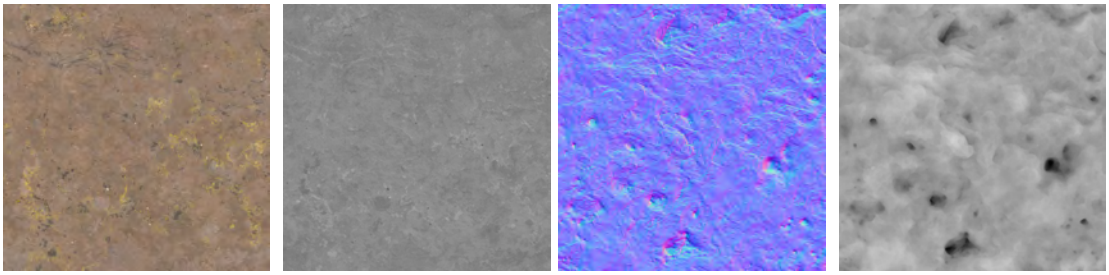
D.3 Snow



(a) Level 1 Normal (b) Level 1 Height (c) Level 2 Normal (d) Level 2 Height

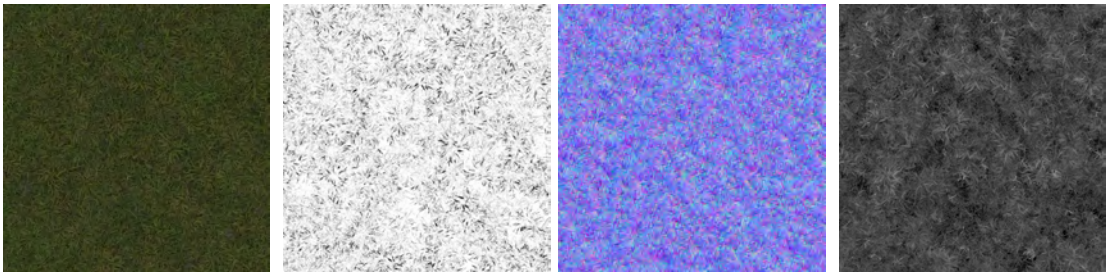
Figure D.6: Level 1 and 2 snow textures.

D.4 Grass



(a) Base color (b) Roughness (c) Normal (d) Height

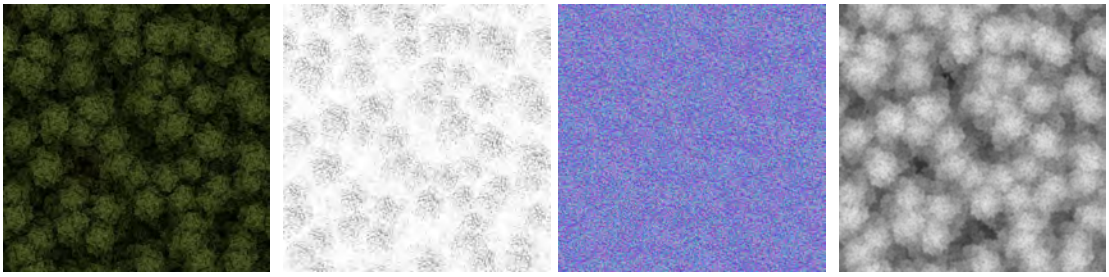
Figure D.7: Level 1 grass textures.



(a) Base color (b) Roughness (c) Normal (d) Height

Figure D.8: Level 2 grass textures.

D.5 Forest



(a) Base color (b) Roughness (c) Normal (d) Height

Figure D.9: Level 1 forest textures.