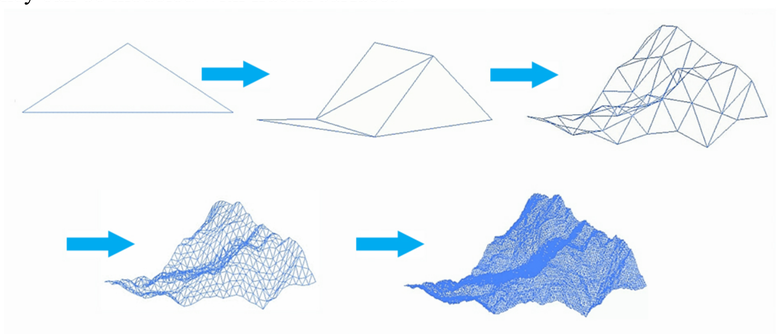
MOTIVATION

Terrain generation is an important usage of computer graphics modeling for industries like video games and 3D movies. In order to render realistic terrain forms on screen, GPUs need to calculate vertices with reasonable height values and handle the shading and display of the terrain. In our experiment, we explored the possibilities of parallization with Cuda in the Square-Diamond algorithm for random terrain generation developed by Loren Carpenter. We chose this topic because we believe rapidly computed vertices for terrains could be very valuable for increasing the performance of rendering process of graphic engines and hence improve the 3D experience for the users by having a higher resolution and more details on the terrain without lags.

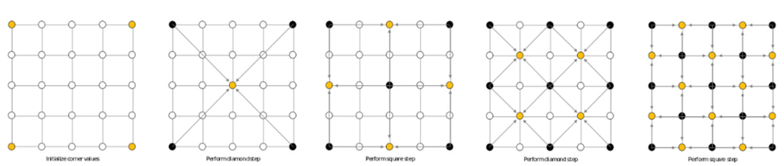
ALGORITHM EXPLANATION

FRACTAL LANDSCAPE: A surface generated with a stochastic algorithm in order to mimic and simulate the appearance of the natural terrain. Therefore, the result exhibits a random behavior and is not deterministic. The basis of this idea is that many natural phenomena shows some patterns that indicate that they can be modeled with fractal surfaces.



DIAMOND-SQUARE ALGORITHM: According to Wikipedia, the Diamond-Square Algorithm is a method for quickly and randomly generating fractal height maps for computer graphics. It was developed by Loren Carpenter, the co-founder and chief scientist at Pixar in 1980s which made the creation of realistic terrain possible. The core idea of this algorithm is to create a smooth looking shape and break in into pieces over and over again.

STEP1: Initialize a 2D array with size of 2^N + 1 by 2^N + 1 to represent the hight-map.
STEP2: Hardcode the value at four corner vertices.
STEP3: For each square on the height map, set the midpoint to:
average of four corner points + some arbitrary random value
STEP4: For each diamond on the height map, set the midpoint to:
average of four corner points + some arbitrary random value
STEP5: Divide the height map into four sub-squares. Repeat STEP4 & STEP5 until all the values on the hiegh map are set once.



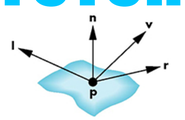
Animated fractal mountain: https://en.wikipedia.org/wiki/Fractal_landscape#/media/File:Animated_fractal_mountain.gif

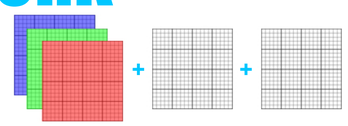
Diamond Square" by Christopher Ewin - Own work. Licensed under CC BY-SA 4.0 via Commons - https://commons.wikimedia.org/wiki/File:Diamond_Square.svg#/media/File:Diamond_Square.svg

TERRAIN Random Generation OPTIMIZATION With & Square ----diamond algorithm

Haoming Lai **Yunhan Ma** **Bangqi Wang**

FUTURE WORK



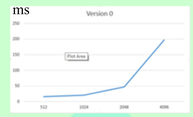


$$I = k_d I_d \, \mathbf{l} \cdot \mathbf{n} + k_s I_s (\mathbf{v} \cdot \mathbf{r})^{\alpha} + k_a I_a$$

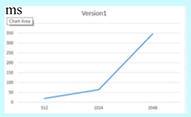
In our project, we simply focused on the possiblity of parallelization for the diamond-square algorithm in order to generate hightmaps for a terrain. It turns out that other parts of terrain generation process could have more potential for parallel optimzation. When terrain is rendered, we let graphic engines such as OpenGL and WebGL to hande the shading for us. We speculate that the calculation process of the Blinn-Phong shading model was implemented within the shader code and the engine will optimize them with parallelization automatically. If we had more time, we would like to explore the possibility of optimizing the shading matrix calculations shown above. For example, we could do tile-based matrix multiplication on each RGB channel with the specular and diffusing coefficient matrix to create shadow and reflection effects on our map.

DATA ANALYSIS

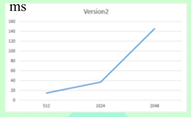
VERSION 0
Serial: 0 kernal, 0 device, no kernel call
This version is simply the linear implementation of the algorithm. No kernel is called in this piece of code and interations are entirely done by the host. it is used as a reference of comparision for other paralleled versions.



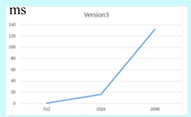
VERSION 1
5 kernels, kernel called iteratively
The iteration process of sub-squares with diamond and square steps will be accomplished by calling a for loop in the host. For each iteration, the program will call 5 kernels. We need to call kernel 5*LogN times in this version.



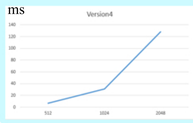
VERSION 2
2 kernels, kernel called iteratively
In this version, the four diamond kernels in Verison 2 is synthesized into one kernel. Other features of the program does not change. By this means, we reduced the amout of total call of kernels to 2*LogN times.



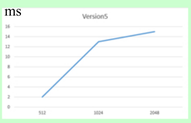
VERSION 3
2 kernels, kernels called iteratively
This version is a modification of Version 2 by letting 1 diamond kernel "smartlly" handle the diamond step's data input and output with different index formula. Each vertex on the heightmap then will be assigned with one thread to handle it.



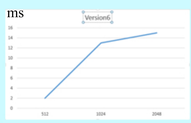
VERSION 4
1 kernel, kernel called iteratively
This version combines the two kernels of Version 2 into one kernel and further reduced the kernel calls in host. The host now only need to call kernel LogN times



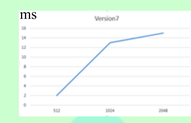
VERSION 5
1 kernel, 2 device, 1 kernel call
This version only calls the kernel once in the host. Two device functions are called to execute diamond and square steps respectively and they are called in the kernel with a for loop iteration.



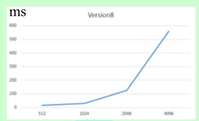
VERSION 6
1 kernel, 2 device, 1 kernel call
During the iteration of our algorithm, the amount of thread actaully needed are contantly increasing by a factor of two. This version only assigns required amount of threads at each step.



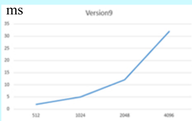
VERSION 7
1 kernel, 2 device, 1 kernel call
In this verison, we manipluate the size of input to kernel during the iteration. In other words, we only call the kernel size we needed with the help of two device functions.



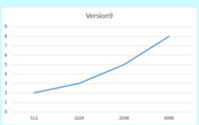
VERSION 8
2 kernel, 2D array used
This is a variation derived from Version 3. Simply manipulated the thread index and call more threads with the 2D implementation.



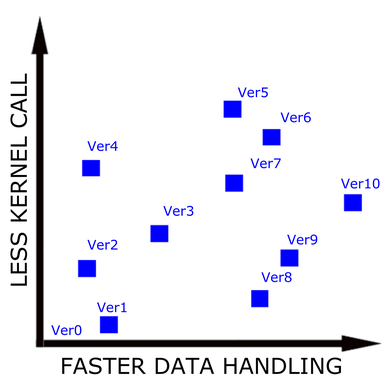
VERSION 9
2 kernel, 2D array
A smarter implementation of 2D array in Version 3 instead of simply adding dimensions.



VERSION 10
2 kernel, 2D array
This version is a combination of all previous successful versions. For each iteration we allocate different size of block according to how many computation we actually need.



OPTIMIZATION APPROACH



AUTHOURS

hlai10@illinois.edu
yma19@illinois.edu
bwang34@illinois.edu

CONCLUSION AND FINAL THOUGHTS

After running all these version with an Intel Core i-7 processor and Nvidia GTX 970 graphic card, we developed version 10 appears to be our best optimization of the square diamond algorithm. For version 10, we combine all improvements from the previous versions. The comparison between version 2 and version 3 indicate a big performance boost by synthesizing 4 separate diamond steps into one. This strategy helps in reduction of divergence because the threads in a thread warp just execute the same calculation processes for different points and they will stay within kernel with assigned indices instead of using condition statements. Then the comparison among version 5, 7, 6 and 2, 3, 4 indicates that the performance optimization results would be much better when the threads are only activated when they are required. In other words, activating all the declared threads all the time would waste many resources and cut the performance. With the experiment in 1, 2, 4, 5 where we tried to reduce the amount of total kernel calls, we found there was only trivial differences. Finally, we combined all the proved usable optimizations in to the final version 10 and this implementation helps us speeding up to 100 times faster than version 0 which is the serial version.