Dynamic Proximity Clouds on the GPU

Ryan Thomas and Sudhanshu Kumar Semwal

Abstract Ray tracing is a widely used technique in rendering realistic scenes in Computer Graphics. Its main drawback has been that it is time consuming, requiring the rendering to finish from hours to sometimes days. For decades, the goal has been to speed up the processing of these scenes. Two popular grid traversal techniques have emerged: (a) Three Dimensional Digital Differential Analyzer (3DDDA) and (b) the Proximity Cloud (PC), which is a variation of 3DDDA. Both of these techniques try to limit the number of collision tests, which can be the most time consuming part of the algorithm. While both techniques allow impressive speedups, large dynamical varying scenes topology still challenge the real time rendering process. These techniques are optimal on static scenes, but object movement forces recalculation of the scene. This is a problem when using CPUs because parallelization is not easily available. Running these on GPUs, however, allows for parallelization. Apart from briefly summarizing some of our previous results from Ryan and Semwal (Proceedings of the world congress on engineering and computer science 2014, San Francisco, pp. 376–381 [1]), we also look to answer some of the more relevant questions about ray tracing, and what future holds for this area.

Keywords Acceleration · CUDA · GPU · Graphics · Grids · Parallelization · Raytracing

R. Thomas (✉) · S.K. Semwal
University of Colorado at Colorado Springs, 1420 Austin Bluffs Pkwy Colorado Springs, Colorado Springs, CO 80918, USA
e-mail: rythomas@gmail.com

S.K. Semwal
e-mail: ssemwal@uccs.edu

© Springer Science+Business Media Dordrecht 2015
H.K. Kim et al. (eds.), *Transactions on Engineering Technologies*,
DOI 10.1007/978-94-017-7236-5_20

1 Grid Acceleration Techniques

The 3DDDA algorithm is a method that works with grod of voxels. These voxels are populated with the objects in the scene and allow the ray to test only objects in the voxel the ray is currently in. The 3DDDA method removes many unnecessary collision detection tests because we test only those objects which are along the path of the ray. The PC method builds from 3DDDA. PCs allow for a ray to skip a larger portion of the grid by computing how far the ray can safely jump before it might have a collision. Both methods are summarized below and are also explained in this paper [1].

2 3DDDA

Fujimoto, Tanaka, and Iwata proposed the 3DDDA algorithm in 1986 [2]. The algorithm defines a grid of voxels and determines which objects inside the voxel need to be tested against. This method would divide the scene into a reasonable sized grid. This grid is used to speed up the ray tracing process. When a ray is fired, it detects whether it hit the grid. If the grid is missed, the ray reflects the background color back showing that no object can be seen from the eye at that position. If the ray hits the grid, the cell (voxel) the ray hits is determined. The traditional ray tracing algorithm is then run on objects that are contained in the current voxel. If no objects are hit in the voxel, the ray then traverses the next voxel on the grid and continues to test each voxel in the path of the ray. This allows for a ray to only perform collision detection on objects that are in its path rather than every object in the scene, as also explained in [1].

The size of the grid does matter, but it is usually not clear what the optimal size of the grid should be in the sense of reducing the image generation time. There is a tradeoff between memory and speedup. The smaller the voxels the fewer the objects a ray will have to test against. If a scene is represented by four voxels then it is still possible that lot of unnecessary collision calculations would have to be computed, but the memory footprint of voxels would be small. If the grid represents a single $1 \times 1 \times 1$ unit in space, then the scene can truly optimize the number of collision detections but would have a larger memory footprint. The size of the grid needs to be determined by the user because the hardware may not be able to support the required memory needed to hold it. This 3DDDA algorithm also has the possibility of being optimized by testing fewer voxels if they are empty, which allows for skipping in the algorithm. There is the potential a large number of grid cells in a row can be empty. This observation is the basis for the Proximity Cloud method, see also [1].

2.1 Proximity Clouds

In 1994 Cohen and Sheffer [3] proposed another grid traversal technique to speed up the ray tracing algorithm. Similar to the 3DDDA method, the scene is divided into many voxels. Once divided the grid cells determine a safe distance a ray may skip between voxels. This results in faster traversal of the voxel grid. When a ray hits a voxel, it first determines if any objects need to be tested. If the voxel is empty or the ray did not intersect with any object, the ray skips ahead by the value the voxel says is safe to skip. This allows for fewer calculations when traversing the grid. There are a few ways this can be accomplished. Using distance transforms, these safe values are calculated during preprocessing.

A method that determines a safe distance to skip is the minimum Euclidian distance between a voxel and all of the non-empty voxels. This would give the most accurate reading for a safe distance. A more optimized method is the city block distance method. The city block distance is calculated by finding the distance in x, y, and z direction of the current voxel to a non-empty voxel and calculating it. This is performed for all of the voxels in the grid. This is not as accurate as the Euclidian distance, but it lets the program avoid calling the square root function on each of the cells. In order to compensate for the possibility of overshooting the target, the ray is normally brought back a grid cell to ensure that no objects are missed. It then continues after checking for possible intersection with that voxel. In the worst case Proximity Clouds perform the same as the 3DDDA method. This technique works great when a scene is divided across large areas; but for very close groups, the cost of building the cloud may not justify the traditional 3DDDA method. This is also explained in [1].

2.2 Directed Safe Zones

The Directed Safe Zones method was developed in 1997 and was an addition to Proximity Clouds [4]. A Proximity Cloud, as described above, finds the minimum safe distance that can be skipped by the ray as it leaves the voxel. This does not take into account the direction that the ray is traveling in. The Directed Safe Zone method takes this into account by leveraging the knowledge it has about the rays direction. When computing the clouds, the method finds the safest distance for each face of the voxel, allowing the rays to skip a larger distance depending upon which face it emerges from. This requires six times the memory, one for every face, to store the variables for the skip in each direction. When a ray leaves a voxel, the face it emerges from is determined. A lookup is performed to find the minimum safe distance the ray can travel based on the face that the ray emerges. This allows the ray to skip an even greater distance because there could be more empty space on one side of a voxel than the other. The performance of this method at its worst is again equal to the 3DDDA method because the minimum safe distance in all

directions is the proximity cloud distance. Similar to Proximity Clouds, the goal of Directed Safe Zones is to travel through the scene at a faster rate. Also see [1] for more explanation.

2.3 Slicing Extent Technique

A different approach to the grid is the Slicing Extent Technique (SET) developed in 1987 [5]. The SET method projects three-dimensional space into two so the grid method is done on a 2D plane. The slicing is done by taking perpendicular slices to the x-axis, perpendicular to the y, and perpendicular to z. Each two dimensional slice is then divided into cells. The ray traversal occurs when the ray moves from one two dimensional cell to another along the path of the ray.

2.4 Modified Slicing Extent Technique—MSET

The MSET method is an extension of the SET method for dividing up rays [5]. The SET method has a couple of shortcomings that the MSET improved upon. The first is slices are proportional to the number of objects. This would result in an unmanageable amount of slices, which made traversal time consuming. The second was the usage of floating point operations to traverse the slices. MSET evenly spaces the slices along the axis similar to the 3DDDA method. This allows for faster 3DDDA grid traversal. MSET takes into account the direction of the ray as well. When the object list inside of a cell is built, it breaks them up based on what direction the ray can hit it. The example is rays traveling up and down through the cell can only intersect with objects above and below the cell, not left and right. This method's ability to predict the size of the data structure will make it ideal for porting to the GPU in the future. The Directed Safe Zones, Slicing Extent Technique, Dual Extent, and MSET methods were not tested in this paper but warrant further research in the future, because in our experiments, as described in [4], DSZ methods outperformed the Proximity Clouds method, based on both experiments and theoretical analysis.

2.5 GPU Computing

Graphical Processing Units have become common in most computers today. They have been around since the 1980s with the goal of speeding up a rendering process for the system. This allows for better processing by freeing up the CPU from rendering by having dedicated hardware to draw to the screens. GPUs are built with many processing cores that run in parallel. The recent trend has been that the GPU

manufacturers are providing APIs for traditional processing on the GPU rather than just graphics rendering. Object hierarchy methods such as k-d tress [6] and BVHs [7] are used for these implementations. But the above mentioned approaches are based on partitioning the object space and may not be suitable for ray tracing a 3D volume data-set, such as that available from CT/MRI slices or even the visible human project. This is because volume data does not have any objects to build the object hierarchies on.

GPU computing has been growing over the previous few years. Supercomputers like performance can be provided on our desktop with multiple GPUs. In our implementation, the 3DDDA algorithm remains the same when ran on the GPU or the CPU. Proximity Clouds were allowed a different approach on the GPU. The principles of the algorithm remain the same but are rewritten using a parallel processing implementation. The benefit is the simplified loop, which allows the distance calculations to be run simultaneously while building the Proximity Clouds. These adjustments allow Dynamic Proximity Clouds implementation.

Modern day CPUs generally have two to four cores on the standard desktops and up to sixteen cores on server CPUs. A device running on the GPU has the potential to have thirty-two threads on a single core. Threading is also made easier in languages such as CUDA. This provides a massive amount of computing power in the average consumer computer. There are three main libraries that allow processing on the GPU: Microsoft's DirectCompute (which is bundled with DirectX 11), OpenCL, and Nvidia's CUDA library. We have used CUDA for our implementation.

The disadvantage which implementations face on the GPU is the memory constraints. For a GPU implementation, usually we are limited to the available memory on the device. This is somehow copied from the host (CPU) memory space, but this operation can be time consuming. This could be tricky when handling dynamics objects and topologies, such as for game applications. In this case, one set of data needs to be exchanged with another modified set. This research does not investigate acceleration techniques to buffer memory, but that would be interesting question for the future [1].

3 Ray Tracing on the GPU

3.1 Implementation

The first step to setup ray tracing implementation on the GPU is to set up and run the traditional algorithms. On the GPU the rays that emanate from the eye are divided into grid segments and run on concurrent threads as described by threading in CUDA. This results in a tremendous speedup from the version on the CPU. The algorithm remains the same for the kernel drawing the spheres [1].

3.2 3DDDA

The 3DDDA method on the GPU is similar to the version on the CPU. A grid is still built in a similar manner. The GPU implementation has the same algorithm when traversing the grid. It does fire many concurrent rays similar to the traditional one. The threads then traverse the global grid. The pre-processing is done in one kernel, and the traversing is done in another. Although explained in more details in [1], here we briefly describe this approach.

The first step is to determine which voxels are filled and which are not. The grid is a set size on the GPU since dynamic allocating and releasing of memory prevents real time rendering. Each voxel is made up of a structure. Each voxel contains a boolean to determine if it contains any spheres, an array of indices for the spheres located in the voxel, and three integers describing the x, y, and z position. This helps in the parallelization since the voxel array is declared as a single dimension array.

Building the grid is done with two separate kernels. The first kernel is designed to empty the grid. This is needed to register spheres with the correct grid cell. This is threaded in the x and y directions. Each thread loops over the z direction setting each cell to empty and resetting the array value to -1, which represents the end of the list of spheres contained in the voxel. This array is static because the GPU prefers static memory to dynamic. This can become a limitation of the system; however, when looking at enough spheres to fill a single voxel other memory issues may be introduced. In that case a new grid of a different size can be built or that array can be increased. On certain cards the memory limitations may be met building the grid and should be taken into account.

Once the grid is reset, it needs to be populated with the spheres contained in the spheres array defined using a float3 data type. The float3 data type is defined in CUDA and contains three floats. The min and max of the sphere is determined in order to create a bounding box used to fill the voxels. Spheres also define a material, which determines their reflectiveness and color. An array on the GPU defines the spheres that are to be rendered. This global array is used to populate the voxels in a second kernel. This kernel is a single dimensional kernel that threads on the number of spheres. Each sphere fills the cells that are contained within its bounding box. Once this kernel is finished the grid traversal can begin.

The final kernel in the process draws the scene. It is broken up in the same way as traditional ray tracing, but it does not compute the collision for each sphere, rather traversing the grid in a device function as explained in [1].

3.3 GPU Proximity Clouds

Proximity Clouds was built using a variety of distance computations. All of them require at least an n squared algorithm to compute, usually two pass algorithm as

described in [3] is implemented. These algorithms can be parallelized to run on the GPU. The algorithm presented in this paper is as follow:

- The index represents the voxel that needs to determine its distance.
- The thread in the y direction represents a voxel that is not empty.
- If a thread receives a voxel that is empty, the thread returns and asks for the next voxel.
- Once all threads in a block are completed a new block is loaded on the GPU.

This builds the clouds in a way that can be scaled to multiple GPUs and can run on any GPU. It is possible to run this in a single warp when sufficient threads are available. This algorithm can take any distance computation as input to determine a safe distance to jump. The speed of this algorithm comes from the parallelization of the system. The first list is divided up into many different threads so they can run in sync. The Proximity Cloud generated for the experiment consisted of a 40 × 40 × 40 grid. The list is 64,000 in length and the second list would be of equal size for the worst case, as also explained in [1].

4 Results

The results were run on a Windows 7, 64 bit PC with 6 GB of RAM, an Intel i7 2.66 GHz processor, and an NVidia GTX 570 graphics card. The GTX 570 has 480 CUDA cores with a graphics clock of 732 MHz and a processor clock of 1464 MHz [1]. Figure 1 shows a variety of techniques we have developed and their relationships with the other existing techniques.

4.1 Clustered Spheres Performance

We started with a rendering of up to 15,000 spheres and a 20 × 20 × 20 grid. This grid size is important because sometimes larger grid side could lead to time out when filling the grid, as will be discussed later. This scene has the quality that the scene had no real gaps. So we expected that the cloud and the grid would perform at the same speed. The proximity cloud only produced a 0.3 % increase in speed, but it did speed up traditional ray tracing by 91.8 %. For more discussion, please refers to [1].

4.2 Clustered Spheres with Large Grid

To better demonstrate the performance of a larger grid, the same scene as above was generated using a 40 × 40 × 40 grid. Since larger grid size was used, we needed to reduce the number of spheres. Far fewer spheres were used due to the timeout that

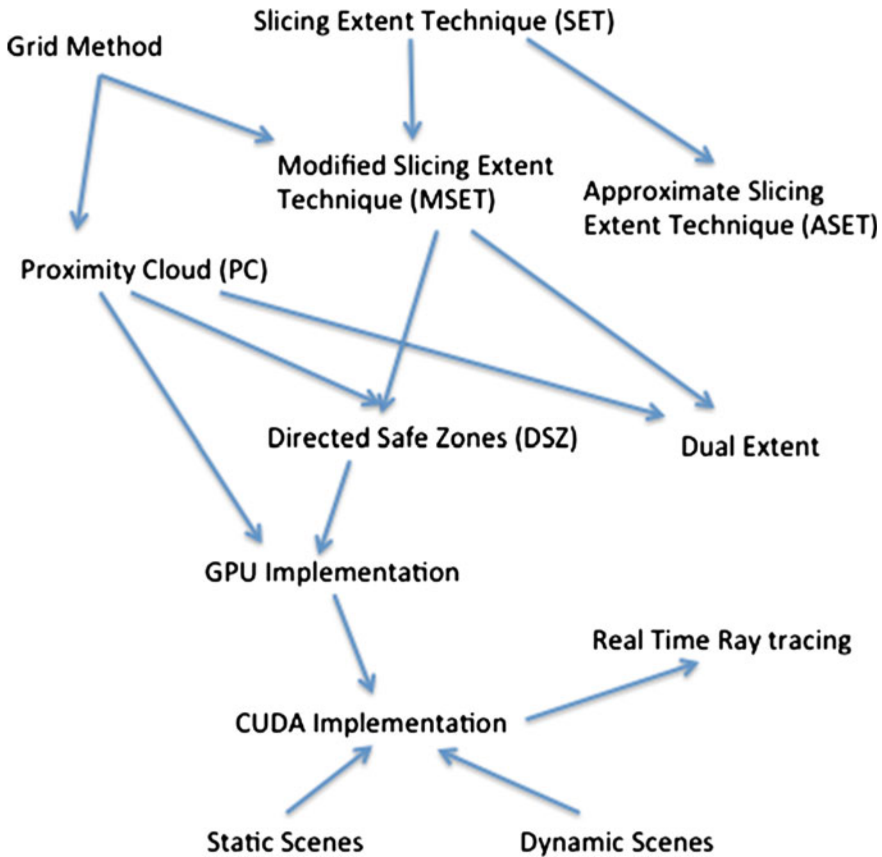

Fig. 1 GPU implementation towards possible real time rendering

can occur when building the Proximity Clouds with the addition of rendering. [1] Once again there is not a huge distinction between the grid and the Proximity Cloud method. This is because there are no large empty areas. There is a 6 % increase in speed when looking at proximity clouds over the grid method, and a 79 % increase in speed over traditional ray tracing.

4.3 Rendering Planes of Spheres

In order to showcase the advantage that has been claimed for Proximity Clouds, a new scene was generated where two planes of spheres are created. One plane is toward the front of the scene, while the second plane is far at the back. This allows for a large gap between them demonstrating how Proximity Clouds increases

rendering time. The grid and Proximity Cloud methods show much faster performance over non optimized ray tracing. The difference between proximity clouds and the 3DDDA method is not as large as expected. This is partially due to the size of the grid. A 40 × 40 × 40 grid can only skip the length of the grid in the best case. This does not allow for huge performance increases when each thread is traversing the grid simultaneously. The average increase in speed was only 6.4 % when looking at all points on the graph, but a 94.2 % increase from the non-optimized ray tracer was achieved. As the scene grows and becomes sparser this should only increase as also indicated in our paper [1].

4.4 Proximity Cloud Generation Speed

The next set of data that was looked at is the speed at which Proximity Clouds can be generated. This is based on the size of the Proximity Cloud as well as how populated the scene is. Each test below generates spheres in random locations and tests the speed at which the Proximity Clouds can be generated. When building a 10 × 10 × 10 grid, it can be put in a single warp. Building it only takes 1 ms, which is a single loop through the scene. A 20 × 20 × 20 cloud is a little bigger, but it experiences a similar behavior to the 10 × 10 × 10 cloud. Once at 350 ms it begins to level out no matter the number of spheres added because the scene has a sphere in every voxel in our implementation for this case.

While slower the 30 × 30 × 30 grid is still manageable. Similar to the 20 × 20 × 20 grid, the 30 × 30 × 30 builds in the same time scale. This is due to a large number of CUDA cores working in parallel. The 30 × 30 × 30 grid requires the same number of warps as the 20 × 20 × 20, thus completing in the same amount of time.

The 40 × 40 × 40 grid requires multiple loops in order to build. Building the grid takes over 1.4 s, and when built with other kernels, it has the potential of running into the kernel timeout. These results do show that running across multiple cards can increase the speed of Proximity Clouds so that it’s closer to the speed it takes to build the 10 × 10 × 10 grid. The speed to build the Proximity Clouds, plus the time to render, is still faster than the traditional ray tracer and a slight improvement over the 3DDA method [1].

5 Future Work

GPUs have permeated the graphics industry and we expect their usage to only grow more. One area to extend this work is to use some form of Cellular Automata for implementing dynamic changes in the scene because the changes usually are localized. Cellular Automata [8, 9] could implement multi-level interactions and emergence of diseases [10, 11]. Complex Systems science [12] has been applied to

model events occurring in nature. Works by Prigogine [13], in thermodynamics, and earlier work by Poincare's on sensitivity of dynamical systems to initial conditions provide the basis for complex systems research for Cellular Automata research. Limitations of simulating organic life by using computational models have been discussed before, these include (i) brittleness [14] of the computational medium, and (ii) the limitations of reductionist approaches to model organic life, which is well documented in [15]. Because Cellular Automata uses local interactions, not the reductionist approaches, it could provide a suitable platform to model organic behavior such as cancerous growth patterns. Local interactions, usually implemented for every cell, could create subtle interactions mimicking organic behavior. Many examples, such as flocking, and 3D games have shown remarkable variety of emergence when a cell's next state is based on consulting nearby voxels. For example twenty-seven cells could be consulted for ($3 \times 3 \times 3 = 27$; 26 immediate vicinity, and 1 itself) to decide the next state. Different non-linear and dynamics pattern could emerge using different local interactions strategies [16].

Volume Data provides one-to-one correspondence for use by a Cellular Automata. The Visible Human Project supported (1989–2000) by US National Library of Medicine (NLM) provides a detailed volume data of human body. The process created a very detailed database of volume data 1 mm apart for the male cadaver with 1871 slices, which when stacked create a 3D grid of volume. This created 40 GB of static 3D grid data which might have to be ray traced, or variation of ray tracing called ray-casting could be used. 3D Morphing techniques [17], and for medical applications [18, 19] have been implemented using cellular automata on volume data. However, real-time manipulation of such large data is not possible with the computer systems of today, yet GPU computing provides a promising research direction. The rise of GPU computing has been growing over the previous few years. GPU computing allows for parallelization of algorithms when the algorithm allows for it. The 3DDDA traversal algorithm remains the same when run on the GPU and the CPU. Proximity Clouds are allowed a different approach on the GPU. The traditional algorithms proposed can be mapped on the GPU. The principles of the algorithm are the same but now are rewritten using a parallel processing implementation. The benefit is the loop is simplified allowing the distance calculations to be run simultaneously while building the Proximity Clouds.

6 Conclusion

Dynamic Proximity Clouds were achieved by dividing the problem into several kernels and allowing the GPU to compute each section. Each generated scene consisted of building a blank voxel grid, filling the grid with the spheres, computing the Proximity Clouds, rendering, and updating the positions of the objects. This allowed for a nice animation when the number of spheres was manageable, but can quickly become choppy, i.e. non real-time, due to the size of the grid.

On average it took 1.44 s to generate the clouds, which sometimes had the potential to reach the timeout of the GPU. The 3DDDA provided the largest performance gain in terms of overall speed but was slower when it came to rendering using Proximity Clouds. This algorithm can be run across multiple GPUs providing real time rendering, but new ways should be addressed, perhaps increasing the speed across a single GPU. By running it in parallel, it is possible to build the Proximity Clouds in a single cycle. The speed increase between the Proximity Clouds and 3DDDA on the GPU demonstrates that more investigation is needed to come up with better traversal methods. Continued parallelization of the algorithm will only result in a better speedup. Finally, cloud computing could provide better and novel solutions to this interesting problem.

Acknowledgments This paper is an invited Chapter based on our earlier publication [1] at the WCECS 2014 conference. Although we have added several new sections, some remnants of the old paper still might be present as we started with our original submission [1]. Both authors want to thank the WCECS 2014 conference organizers for inviting us to submit this book Chapter.

References

1. Ryan T, Semwal SK (2014) Ray tracing using 3D grid simulations, lecture notes in engineering and computer science. In: Proceedings of the world congress on engineering and computer science WCECS 2014. San Francisco, pp 376–381, 22–24 Oct 2014
2. Fujimoto A, Takayuki T, Kansei I (1986) ARTS: accelerated ray-tracing system. IEEE Comput Graph Appl 6(4):16–26 (print)
3. Cohen D, Sheffer Z (1994) Proximity clouds an acceleration technique for 3D grid traversal. Vis Comput 11(1):27–38
4. Semwal SK, Hakan K (1997) Directed safe zones and the dual extent algorithms for efficient grid traversal during ray tracing. In: Graphics interface 1997, pp 76–87 (print)
5. Semwal SK, Kearney CK, Moshell JM (1993) The slicing extent technique for ray tracing: isolating sparse and dense areas. In: Graphics, design and visualization 1993, pp 115–122 (print)
6. NVIDIA CUDA (2009) C programming best practices guide. In: CUDA programming guide. NVIDIA, Web. 10 Mar 2011
7. Manocha D, Lauterbach C (2006) Ray tracing dynamic science using BVHs. In: SigGraph 2006 presentation, pp 1–47
8. Sarkar P (2000) A brief history of cellular automata. ACM Comput Surv (CSUR) 32(1):80–107
9. Wolfram S, A new kind of science, book on cellular automata. Wolfram Media Company, London, pp 1–849
10. Bezzi M, Modeling evolution and immune system by cellular automata. http://citeseer.nj.nec.com/429312.html
11. Sosic R, Johnson RR (1995) Computational properties of self-reproducing growing automata. BioSystems 36:7–17
12. Melaine M (2009) Complexity: a guided tour. Oxford University Press, Oxford, pp 1–337
13. Prigogine I, From being to becoming, freeman (ISBN 0-7167-1107-9)
14. Ray T (2000) An evolutionary approach to synthetic biology: zen and the art of creating life, Chap. 2. In: Book on best papers from VW98 Paris conference

15. Stephen R, Lessons from the living cell: the limits of reductionism. McGraw Hill, New York, pp 1–300
16. Rabinovich MI, Ezesky AB, Weidman PD (2000) The dynamics of patterns, World Scientific, Singapore, pp 1–324
17. Semwal SK, Chandrashekher K (2005) 3D morphing for volume data. In: The 18th conference in central Europe, on computer graphics, visualization, and computer vision, WSCG 2005 conference, pp 1–7
18. Fang S, Raghavan R, Richtsmeier J (1996) Volume morphing methods for landmark based 3D image deformation. In: SPIE international symposium on medical imaging
19. Forsyth T (2002) Cellular automata for physical modeling. Game Program Gems 3:200–214