USING BLOCKS TO CONSTRUCT 3D SHAPES AND CREATE TRANSFORMATION ANIMATIONS

A Thesis

by

LU LIU

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

December 2006

Major Subject: Visualization Sciences

USING BLOCKS TO CONSTRUCT 3D SHAPES AND CREATE TRANSFORMATION ANIMATIONS

A Thesis

by

LU LIU

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Approved by:

Chair of Committee,	Frederic I. Parke
Committee Members,	Mary Saslow
	John Keyser
Head of Department,	Mark Clayton

December 2006

Major Subject: Visualization Sciences

ABSTRACT

Using Blocks to Construct 3D Shapes and
Create Transformation Animations. (December 2006)
Lu Liu, B.Arch., Tsinghua University
Chair of Advisory Committee: Dr. Frederic I. Parke

The objective of this research is to develop methods by which we can use blocks to approximate the shapes of 3D objects and to generate shape transformation animations. Two graphic tools are developed. One assists the animator in constructing 3D shapes with bricks of different sizes and matching up the different shapes. The other tool helps the animator generate a transformation animation of those bricks. Using polygon shape data, these tools can procedurally place the bricks and control their animation. Several different methods for animation are introduced. Those methods provide different ways to generate animation paths of the blocks. The no path animation and the straight path animation are easy for the animator to create and the animation time is easily controlled. The flocking animation will provide more interesting effect.

ACKNOWLEDGMENTS

I would like to thank my committee chair, Prof. Frederic I. Parke, for his time, patience, guidance and enthusiasm in both my study and thesis writing. I would also like to thank Mary Saslow and John Keyser who served on my committee for being prompt and supportive.

I would like to thank Tatsuya Nakamura and Josh Rowe for generously sharing ideas with me.

An important thank you also goes out to my parents for blessing and supporting me with their love.

TABLE OF CONTENTS

CHAPTER Page

LIST OF FIGURES

FIGURE Page

CHAPTER I
INTRODUCTION

Building blocks are a popular toy around the world. As a kind of building block toy, LEGOs[1] are designed to make or to approximate almost any shape.

During the past decade, building blocks have also become a theme in some computer graphic developments. Some 3D block animations have been created [2]. Computer aided design software for block based structures has been developed by the LEGO company [3]. Using this software, a user can conveniently choose from a variety of LEGO pieces to create a desired shape. In this study, we present a practical method to build three-dimensional shapes using blocks or bricks of different sizes and to create visually interesting animations of the bricks transforming from one shape to another shape.

I.1. Motivations

The motivation for generating these animations is to help people understand how these two shapes are built up from the same group of bricks; and to create interesting visual effects. These methods are also motivated by Lerios's thesis, *Feature-based volume metamorphosis* [4] and crowd animation effects in PIXAR Animation Studio's movie *Finding Nemo* [5]. These methods can easily be used by animators who have an artistic background and have basic skills using MAYA. These methods are affordable for low

---

The journal model is *IEEE Transactions on Visualization and Computer Graphics.*

budget animation projects that require these kinds of block animation effects

I.2. Objectives

The first objective of this work is three-dimensional shape approximation with blocks. A C++ program is developed that will load in two polygonal 3D geometry files and approximate these polygon shapes with groups of bricks. Then the program matches up corresponding bricks in each group one by one; assigning IDs to each brick. When finished matching, the program exports data for all the bricks of each group into an output data file. This file includes the size of each brick and its position in each shape.

The second objective is to generate transforming animation. A program is created to load in the data file from the shape filling program and check if the data is matched. If it is, the program calculates an animation path for each brick; moving each brick from its location in the first shape to its position in the second shape. Then the program exports the animation paths in a format usable in Maya. An animator can then create the final animations using the powerful shading and lighting tools in MAYA.

CHAPTER II
PREVIOUS WORK

This chapter reviews previous work on topics related to this study. Previous research on volume metamorphosis and 2D block placement is briefly discussed as preparation for solving the matching problem. This is followed by a discussion of the flock animation technique, which will be applied to solve for animation paths.

II.1. Volume Metamorphosis

Feature-based 3D volume metamorphosis [4] applies to changing volume-based representations of objects. It is an extension of Beier and Neely's 2D image warping [6] technique into 3D space. The first step is to divide the source and target volumes into several parts. By manually using points, segments, rectangles and boxes as elements, the user builds corresponding element pairs in each volume. The second step is to transform these elements in the source volume, by moving, turning and stretching them to match respectively the position, orientation and size of the corresponding elements in the target volume.

II.2. Previous Studies about 2D Block Placement

Miyata developed a method for generating realistic 2D Texture maps for a stone wall [7]. His method places random sized rectangle stones, working from the bottom of the image to the top of the image. It then fills in small stones for the remaining vacant spaces. This

is a simple and efficient way to place the biggest blocks first and place smaller blocks for the remaining spaces in a desired 2D shape.

J. Legakis, J. Dorsey and S. Gortler developed a pattern generator that gives a solution for applying texture onto 3D objects [8]. This strategy generates a grid on the object's surface and then fills rectangular bricks with different sizes into the grid. This is very similar to the method we use to fill bricks into a volume as discussed later.

II.3. Flock Animation

The aggregate motion of a simulated flock is created by a distributed behavioral model; much like a natural flock. Each flock member chooses its own course. Each simulated member is implemented using physics that controls its motion, and a set of behavior rules that can be specified by the user. The aggregate motion of the simulated flock is the result of the interaction of the relatively simple behaviors of the individual simulated objects. Flock simulation can be applied to particle systems. Some advanced flock systems have been developed to simulate realistic virtual crowds [9][10].

A good example of flock animation for shape transformation is in the PIXAR movie *Finding Nemo* [5]. In this movie, a school of fish was successfully simulated to form the shapes of several different geometries. Figure 1 shows the school of fish transforming from the shape of a swordfish into the shape of a lobster.

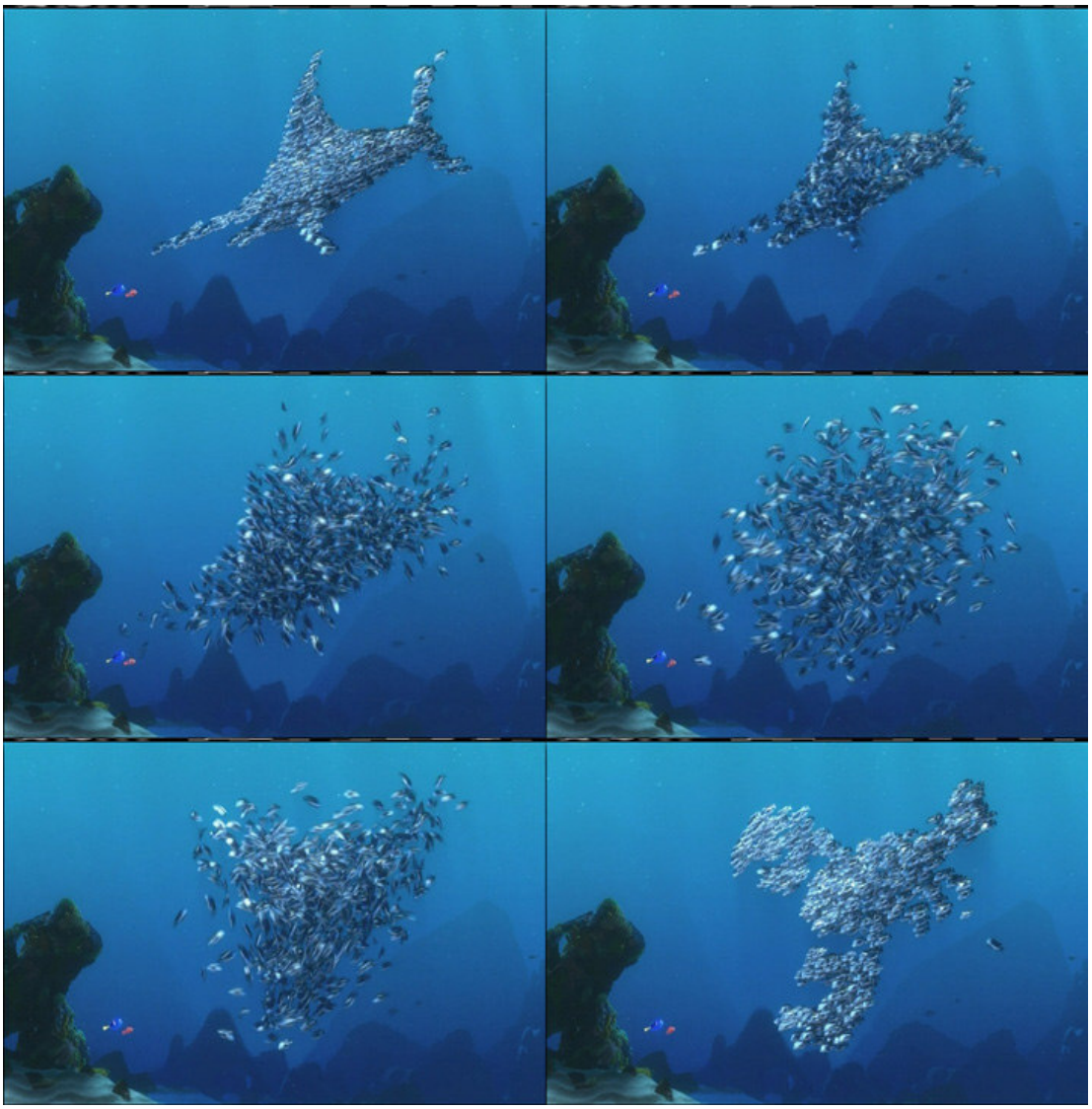

Fig. 1. The transformation of a school of fish.
Images by PIXAR Animation Studios.

CHAPTER III
METHODOLOGY

Figure 2 shows the pipeline of my methodology. I use both MAYA and two programs I developed to get the animation results.

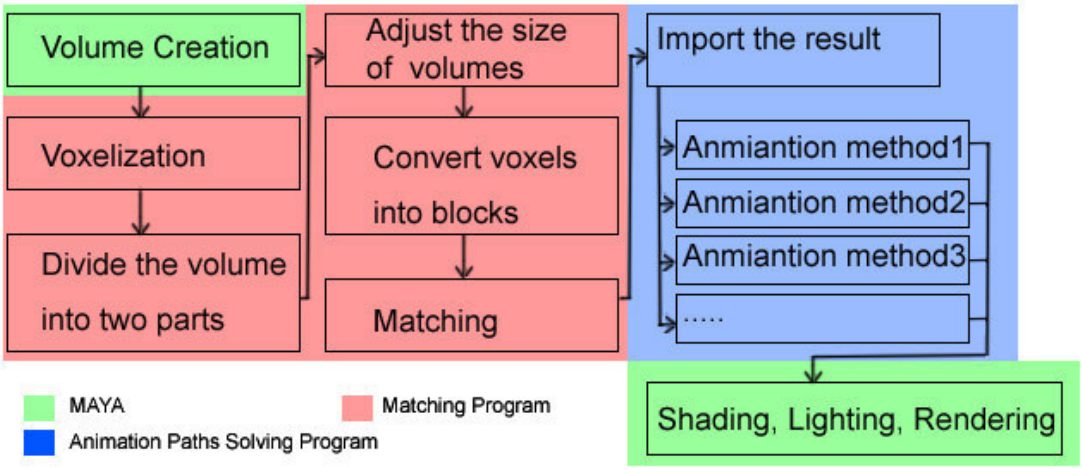

Fig. 2. The pipeline of my methodology.

III.1. Matching Shapes

This section discusses the approaches used to create block approximations to three-dimensional shapes and how the blocks of two shapes are matched. The block matching is necessary to enable the desired transforming animations.

III.1.1. Volume Creation

To create the required volume data to approximate the desired shapes, a program is created to convert the geometric surface shape model into a voxelized model. The first

step in the voxelization is to import a specified polygonal surface geometry. The *.obj* file format is used because it is widely supported by most popular 3D software packages. The imported geometry must be the closed polygonal surface for the desired volume.

After loading the shape data, a 3D grid will be applied to the shape. As shown in Figure 2, the program will send two virtual rays, one up and one down, from each position in the 3D grid. We pick four voxels (A,B,C,D) in the figure 3 as examples. If both rays intersect the geometry's surface an even number of times (as for voxel B and D, the two numbers in the bracket are the number of times each ray intersects the surface), this position in the grid must be outside the geometry. If both rays intersect an odd number of times (as for voxel A and C), the position is inside. If one intersects an even number of times and the other intersects an odd number of times, the geometry is not a closed shape. In this case, we need to fix the polygonal model.

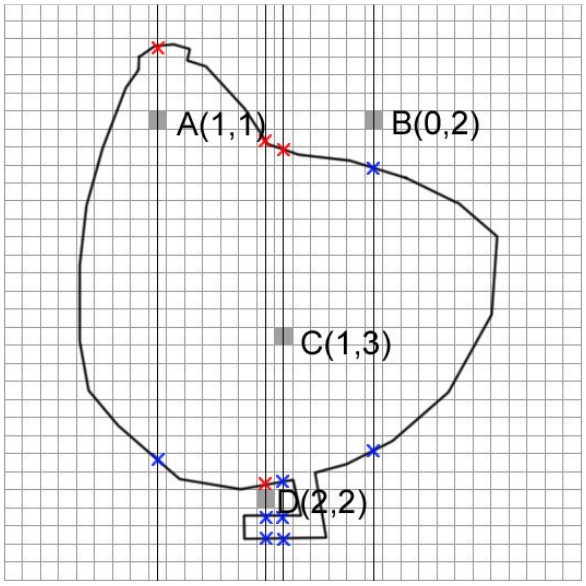

Fig. 3. How to judge whether a voxel is inside the shape or not.

III.1.2. Dividing the Shape into Two Parts

To transform source volume S to target volume T, requires determining a correspondence between the two shapes. Since each shape is formed with different arrangements of bricks, the program must match the bricks of each specific size in the two shapes.

First, we need to match the voxels. In most cases, the number of voxels in S and T are not the same. My solution is to divide the voxelized volumes S and T into surface voxel groups Ss and Ts and interior voxel groups Si and Ti (Figure 4). Assuming a closed shape, we can potentially see all surface voxels, but we can't see the interior voxels. It will be reasonable for the surface groups to have priority in the matching process.

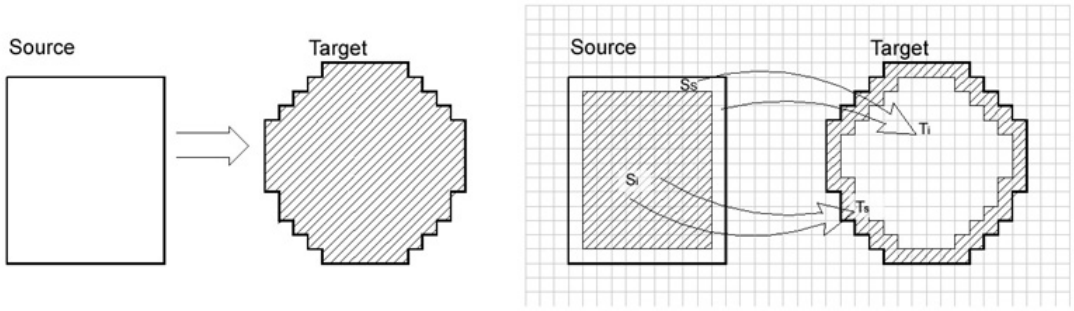

Fig. 4. Dividing each shape into two corresponding groups.

A good solution is to adjust one shape's inside group to match the other shapes surface group. In those cases where we can't match up the entire set of bricks, we can still finish the most important task; to provide completely matched surface groups Ss and Ts.

To achieve this goal, we need to ensure that the inside group of one shape contains no fewer voxels than the outside group of the other shape. If this is not the case, we can simply uniformly scale the shape geometries to enlarge their volumes relative to the 3D voxel grid. The number of voxels inside the shapes will increase cubically while the number of surface voxels will increase by the second power. We can easily find a good scale factor to make the number of inside voxels in each shape larger than the number of surface voxels in the other shape (Figure 5).

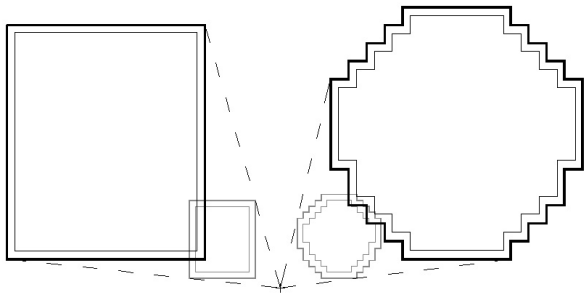

Fig. 5. Scaling the shapes to get enough voxels inside.

III.1.3. Converting the Voxels into Bricks

The bricks we are using to approximate the shape can have different sizes. For this project, the length and width combinations will be defined by the animator, the height of the bricks is fixed to one unit.

The problem of filling the bricks into the modeled shape becomes the problem of packing each layer of the voxelized model with several different size elements (Figure 6). One plausible way in packing is to use bricks as big as possible because it generates groups with fewer bricks. Since there may still be vacant spaces left after we place as

many big blocks as possible, unit size blocks should be used to fill in these spaces.

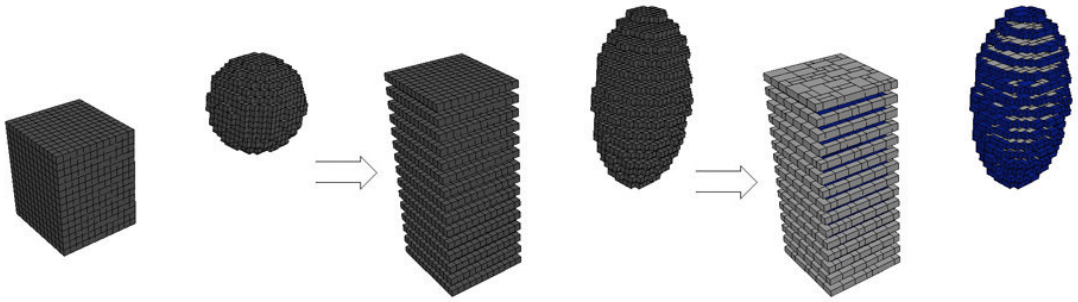

Fig. 6. Filling the bricks into each layer of two volumes.

My method is to define an array *Bsize* which will store the different sizes. In this array, the biggest size will be saved in the first position and the unit size (1*1) will be saved in the last position. If two sizes have the same number of voxels, we can put either one of them earlier in *Bsize*. While converting, the program will randomly pick a voxel in the surface part of the layer and check if it can fill this position with the first size stored in *Bsize*. If any part of the brick is out of the boundary of surface part or intersects with other existing bricks, the program will then use the next size in *Bsize*, until it finds a proper size. The program will keep filling bricks in another randomly picked position until there is no vacant space left in the surface part of this layer. The same method and size array will be used to covert the surface and inside parts into bricks in every layer, in each volume. In Figure 7, I choose two shapes and use one layer in each shape as an example (Figure 7.a). Then I use bricks with sizes of 1*3, 3*1, 1*2, 2*1 and 1*1 to fill in the surface part of cubical shape and the inside part of spherical shape (Figure 7.b).

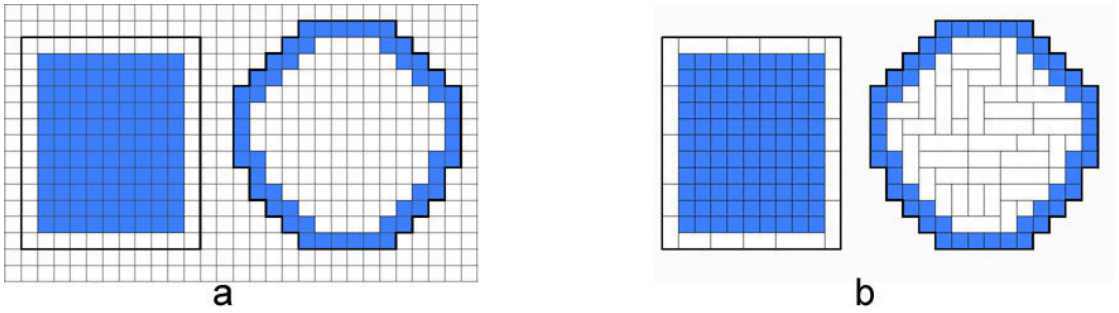

Fig. 7. Converting the voxels in the surface part of the cubical shape and the inside part of the spherical shape into bricks.

III.1.4. Matching

We then want to fill the volumes with groups of different size bricks. For example, the animator defines *n* different sizes. For each group of bricks of the same size in surface group Ss, we label them as Ss1, Ss2 … Ssn. Also, we can use Si1, Si2 ... Sin for labeling the inside group Si. It is the same case for volume T. After we subdivide the volume elements into these secondary groups, we need to match them up, in each shape, one by one. Ss1 needs to match with Ti1, Ss2 needs to match with Ti2 and so on.

To match the numbers of each size of bricks, a good way is to match the number of the biggest size bricks first. If the numbers of the biggest size bricks do not match, extra big bricks can be divided into smaller bricks. On the other hand, two smaller bricks may not be combined into a bigger brick if they don't have the proper size and position. Then we will match the numbers of the second to biggest bricks, down to the smallest size bricks (Figure 8.a to Figure 8.h).

While matching the bricks with the same size, the positions of bricks in each shape

will also be considered. Based on the animation rules discussed in a later chapter, the bricks in the top layer of the source volume will animate first. It will also be good for the bricks in the bottom layers of the target volume to finish their animation as early as possible. For this objective, my method will order the bricks with the same size from the top layer to the bottom layer for each volume. Then it will match the first brick in the array of the source volume to the last one in the array of the target volume, then the second to the second to last, until bricks in one array or both arrays are all matched.

Finally we may have some smallest size bricks remaining. As we previously discussed, these should be unit size bricks. Since the number of voxels of each inside group is larger than the number of the surface group of the other shape, there are some extra unit size bricks inside both shapes. The simplest method is to delete them, which will create some holes in both volumes. A better solution will be to match those bricks from the top of the source volume to the bottom of the target volume, as many as possible. Then the program deletes those brick which can't correspond in the other shape in the final step (Figure 8.i and Figure 8.j).

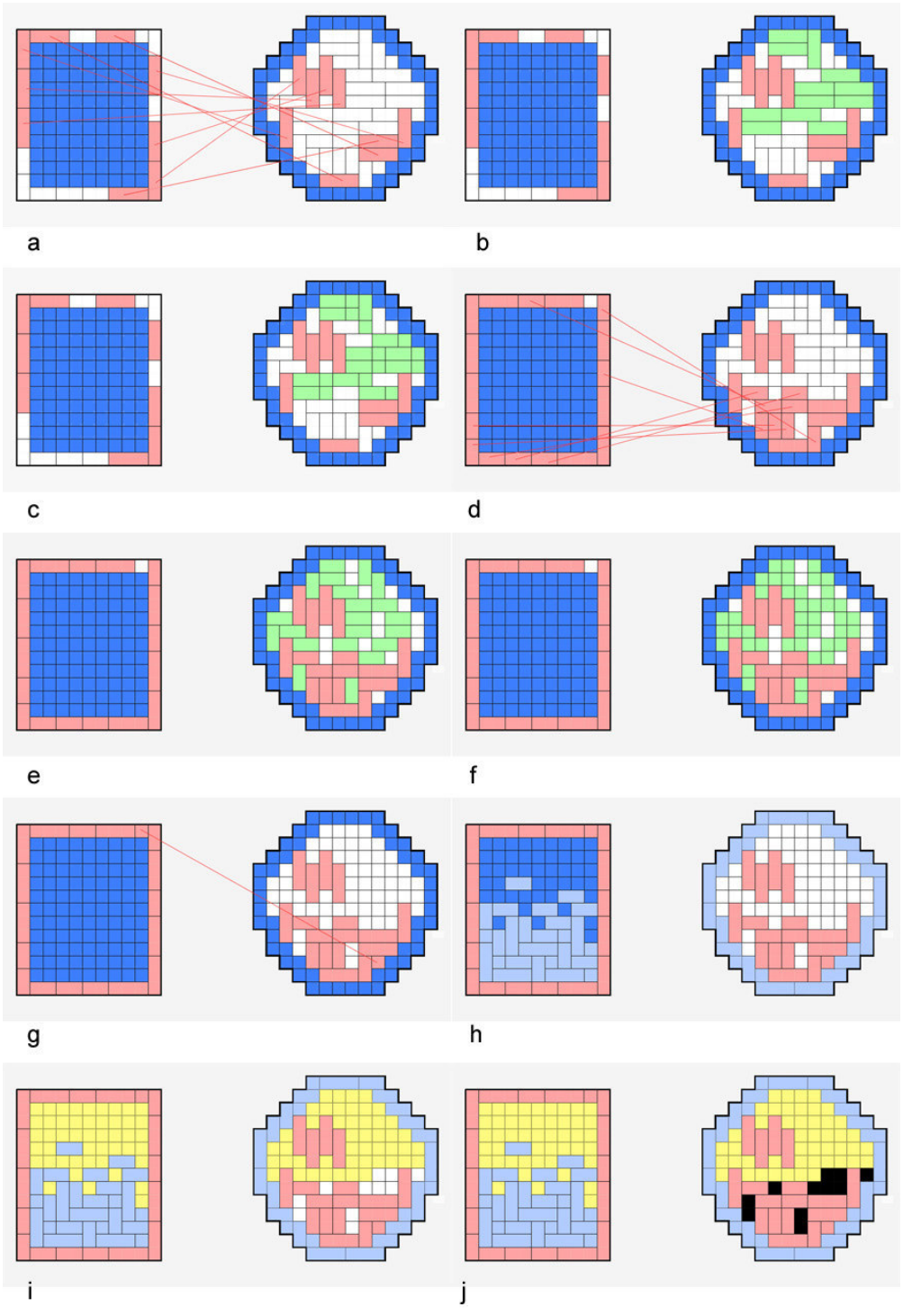

Fig. 8. Matching the bricks in two shapes.

III.2. Finding Animation Paths

The motivation for the generated animations is to help people understand how these two shapes are built up from the same group of bricks and to create interesting visual effects.

III.2.1. Animation without Paths

The simplest method for generating animation is to move each brick directly from its position in one shape (initial position) to its position in the other shape (end position) in one frame time. The sequence of the bricks to be animated is determined by their positions in the source shape. Generally, the bricks will be animated from the top layer to the bottom layer in the source shape. In order to avoid all bricks in the same layer moving at the same frame, the frames for them to be animated are randomly picked in a certain range. For example, the time for bricks in the top layer to be animated will be picked from frame 0 to frame 10, the frame range for the bricks in the second to top layer will be frame 5 to frame 15, then next layer will be frame 10 to 20… This effect looks similar to traditional stop motion animation. It is not necessary to generate animation paths for the bricks.

This kind of animation will work well in some cases that have a small number of bricks. But, it won't let people realize the bricks in both shapes are matched, one by one, if there are several bricks moving at the same time.

III.2.2. Animation with Straight Paths

Another approach is to create an animation path for each brick. Given the initial position

and end position of each brick, there will be many possible paths for the movement. The simplest way is to make a straight path between each position pair. This solution allows good control of animation time. Each brick moves along its straight path in a number of frames. Shortcomings are that there may be collisions between bricks and that the resulting visual effects may not be interesting.

III.2.3. Applying Flock Animation to Bricks

A more interesting solution for animation is to apply flock simulation to the bricks. This approach adds behavior rules to the bricks, such as the collision avoidance behavior described by Reynolds [11]. The set of behavior rules used will determine each brick's acceleration value at each simulation time step.

The set of rules used is:

1. Activation: A brick will not be activated until the bricks above it in the source shape are activated.
2. Collision Avoidance: avoid collisions with nearby bricks. As shown in Figure 9, if the distance h between two bricks P1 and P2 is less that a preset value, those two bricks will accelerate away from each other. In the preset range, the magnitude of the acceleration is an inverse function to h.

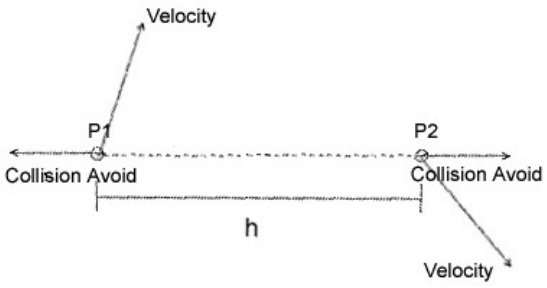

Fig. 9. Collision avoidance.

3. Target Aiming: Attempt to get to the goal position assigned by the matching algorithm by accelerating toward it (Figure 10). The smaller the distance to the target *h* is, the smaller the acceleration toward the target will be. In my implementation, it's a direct ratio. When the brick reaches the target position, the acceleration will be zero.

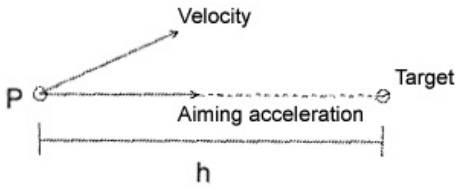

Fig. 10. Target aiming.

4. Deactivation: When a brick is within to a certain distance of its final position and the bricks under it are all deactivated, it will decelerate toward that position and be less affected by other bricks. This deceleration will reduce the brick's velocity based on how close it will be to the goal position at the next simulation time step.

The velocity of each brick will be zero when it reaches its final position.

There are some details on how to activate and how to deactivate a brick. To deconstruct a model of stacked bricks, usually people will take the bricks from the top to the bottom. That is the basic idea we use to activate bricks. This will avoid moving the bricks inside the shape too early; avoiding intersection with unactivated bricks above it.

The simplest solution is to activate all bricks in each layer at the same time. The bricks in the top layer will be activated first, then the bricks in the next layer in the next frame time. Two shortcomings of this method are obvious. First, the motions of bricks in each layer are unified when they are activated. The animation effect is a lack of randomness that will make the animation uninteresting. The second shortcoming will be noticed if the shape doesn't have many layers, for example, if the shape has 10 layers. The bricks in top layer will be activated at the first fame time. The bricks in bottom layer will start their movement at the tenth frame. Standard video has 30 frames per second. The time difference of 10 frames will be 1/3 second. This is a very small amount of time for people to perceive. The resulting animation looks as though all bricks are activated at the same time.

A better solution is to use a randomness factor for the time to activate each brick while making sure the brick will not be activated if there are unactivated bricks above it.

While running, the path generating program will check each unactivated brick to see if all bricks above it are activated. If so, this brick is ready to be activated. The program will randomly choose several bricks from these ready bricks and initialize their motions to activate them. The remaining bricks will still be unactivated and wait to be chosen at later frame times. Figure 11 shows an example of activating a group bricks. The yellow means the brick could be activated, the red means the brick is selected by the program and will be activated in the next frame.

To deactivate a brick, the program must drive the brick to reach its final position and make its velocity and acceleration be zero at that time.

As shown in Figure 12, if the distance, *h*, from the brick's current position, *P*, to its final position, *A*, is smaller than a certain value r, the program will divide the velocity, *V*, of the brick into two parts. One part, *V1*, is aimed toward the final target. The other part, *V2*, is perpendicular to *V1*. Also, bricks within distance *r* of their final position will not be affected by other bricks.

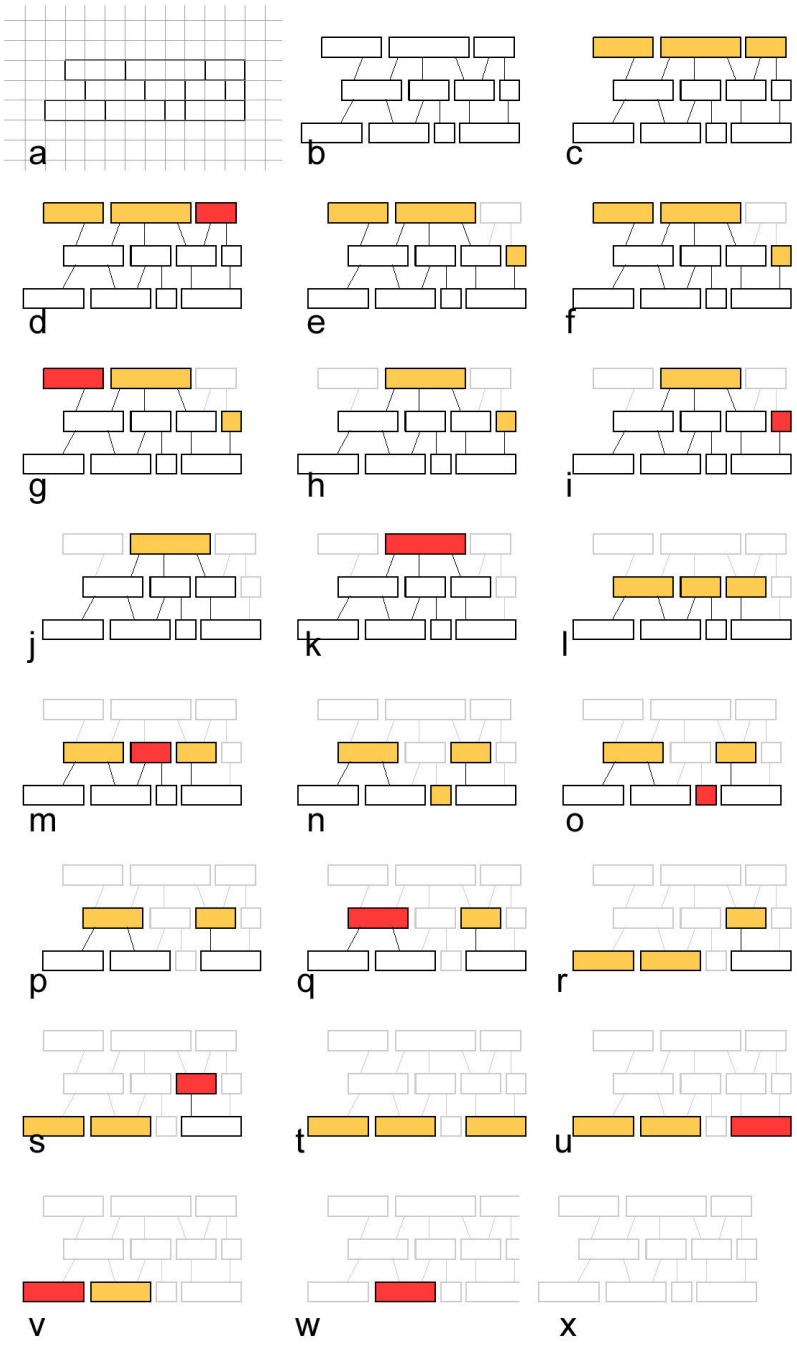

Fig. 11. Activating a group of bricks.

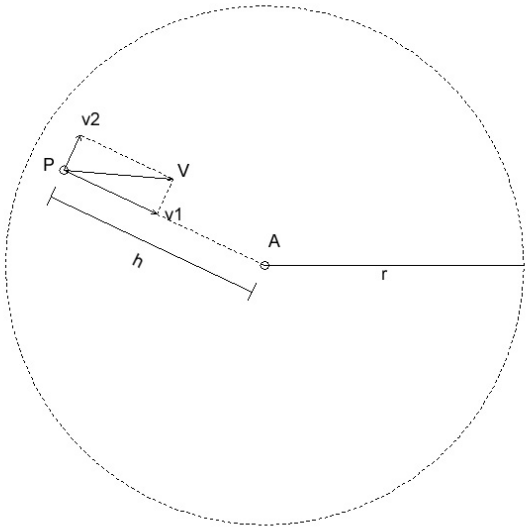

Fig. 12. Brick P in deactivation region.

As shown in figure 13, to deactivate the brick in position P, the program will generate acceleration in both directions *V1* and *V2*. For the *V2* direction, the acceleration will be *n2*V2*, where *n2* is a negative real number set by the animator. This means the brick will get an acceleration opposite to the *V2* direction. The larger *V2*, the larger the deceleration the brick gets. The acceleration in the *V1* direction will be *n1*V1 + Aaim*. *n1* is also a user set negative real number. *Aaim* is the aiming acceleration. The smaller h, the smaller *Aaim* will be, as we described above. By choosing proper values for *n1* and *n2*, the brick will decelerate while moving toward the target. In each next simulation frame time, the brick moves to a new position P' and gets a new position with a new velocity. When h is less than a small threshold value, which means the brick is very close to the target point *A*, the program will just set the position of the brick to *A* and set

the velocity and acceleration to be zero. The brick is deactivated after this frame.

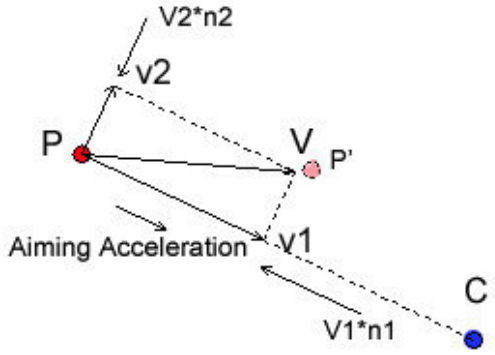

Fig. 13. Method for deactivating the brick.

III.2.4. Assigning Another Position as an Intermediate Aiming Target

Another plausible solution is to procedurally compute an "intermediate aiming target" for each brick in the simulation. The activated brick will check to see if the bricks under it in the target shape are all deactivated. If not, it will aim for its intermediate aiming target and move toward it. When the bricks under it in the target shape are all deactivated, it will switch its aiming target back to its position in the target shape. Then the brick will move toward its final position and get deactivated when it arrives there.

The intermediate position could be randomly generated or generated using certain rules. Different ways for choosing these positions will create different visual effects. Several possible ways could be:

A. Picking positions on a hemisphere surface. This hemisphere is above the two shapes and will cover them like the sky. This is a good choice if the

animator wants to create animation with the first shape deconstructed from the top layer down and the second shape to be constructed from the bottom layer up.

To choose the intermediate positions from the surface of a hemisphere, my method will first create a line between the initial position, P1, and the end position, P2. Then a ray from the center of the hemisphere, C, to the mid point of the line is created. This ray will intersect the surface of the hemisphere. This intersection point, p, will be used as the intermediate aiming target point (Figure 14).

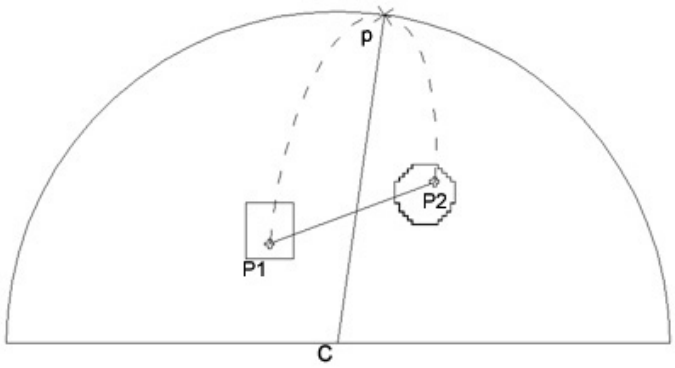

Fig. 14. Picking the intermediate aiming position from the surface of a hemisphere.

B. Picking positions on a spherical surface large enough to include both shapes inside. In this case, the bricks will pop out from the first shape and then converge to the second shape. The movement of the bricks will look similar to

the movement of the school of fish in the film *Finding Nemo* [5] (Figure 1). The better sequence to activate the bricks in this case will be to activate the bricks from the outside to the inside (Figure 15). The intermediate positions will be determined by a method similar to the one we used for picking from hemisphere.

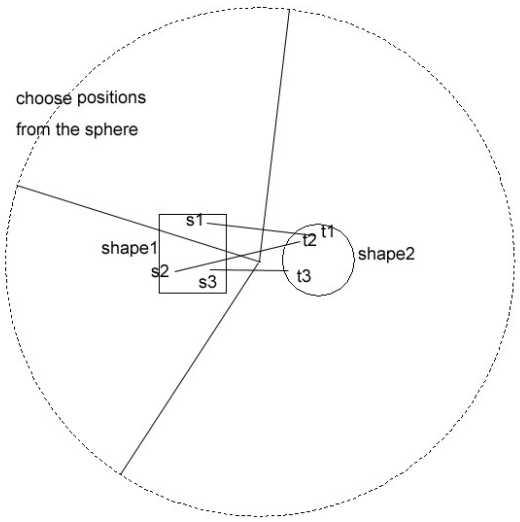

Fig. 15. Picking the intermediate aiming position from the surface of a sphere.

C. Other ways to pick intermediate positions include, for example, picking positions on a plane or picking positions along a 3D curve. In addition to using a single geometry or curve, it is also possible to pick positions inside the volume of some implicit surface.

By assigning intermediate targets to each brick, even if two shapes are very close together or overlap each other, the bricks will still find a position to move toward and wait for the proper time to move to their final positions.

III.3. Exporting the Animation Paths to MAYA

The computed animation paths will be exported into MAYA. During the path simulation, the program will generate one text file for every frame. These files record the IDs and positions of each brick in each frame. The file generated at the first frame includes brick size information.

A MEL script loads in the data from these files, generates the display geometry for bricks with the sizes recorded in the first file, and moves the bricks to their positions in each frame.

The matching program doesn't consider brick color. All bricks created in MAYA are assigned a default shader. The animator can assign the different colors he want to the bricks. It could be one color or several colors for the bricks in each surface. Using MAYA's lighting and rendering tools, the final high quality brick shape transformation animation can be rendered.

CHAPTER IV
IMPLEMENTATION AND RESULTS

IV.1. Implementation

This chapter discusses the implementation of the ideas and concepts presented in the previous chapter. It also discusses the results achieved. The chapter begins with some tips for creating the object models to be used with these techniques.

IV.1.1. Tips for Modeling in MAYA

It's not necessary and perhaps not wise to create very detailed models. The details which are smaller than one voxel will be ignored after voxelizing.

For shape transformation animation, I would recommend avoiding thin flat shapes such as flags or pieces of paper. If the thickness of the shape is smaller than one unit, it won't even generate voxels. A good shape for transformation should have some interior volume. The reason for this has been described in Section III.1.3 .

The positions of the two shapes in space will affect our choice of animation effects. If we want to use effects without choosing intermediate positions for bricks, we should keep the two shapes some distance apart, or at least avoid overlapping them in space.

IV.1.2. The Program for Matching

The program for brick matching is written in C++. Each brick is represented as a data node with ID, size and position attributes.

To help the animator check progress, the program generates a screen window to show the result of voxelizing imported polygonal surface models. Figure 16 shows two example models built in MAYA. Figure 17 shows the window created by the matching program.

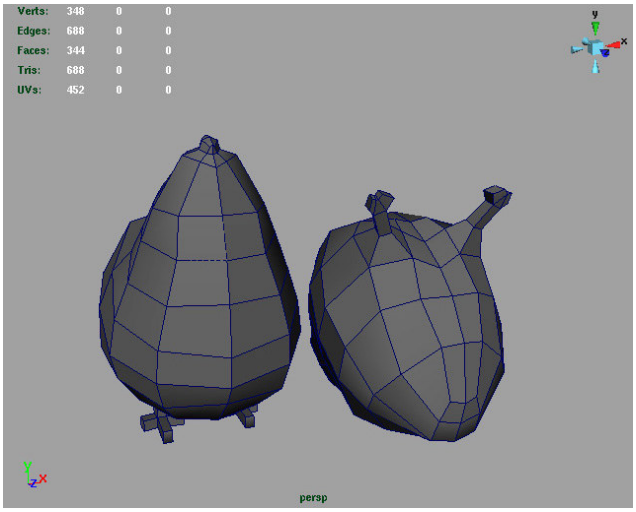

Fig. 16. Two polygonal models created in MAYA.

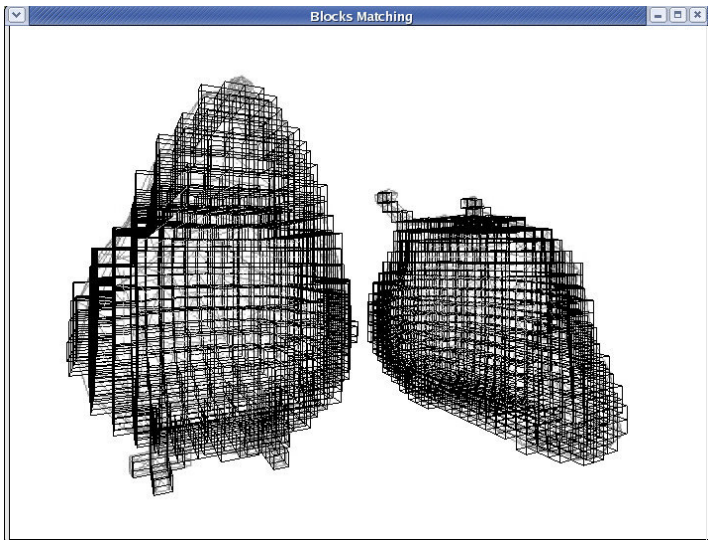

Fig. 17. Polygon models are imported into the program and voxelized.

This view can be interactively controlled by the mouse. The right mouse button controls camera zoom. The left mouse button controls the rotation of the camera. This is similar to the default viewing controls in MAYA.

IV.1.3. The Program for Animation Paths Solving

The animation paths solving program is also written in C++. Each brick is a node in this program. The program uses these nodes to build the flocking system to simulate the motion paths. The program will load in a parameter file created by the animator. The values set in this file will control the effect of output animation.

In most cases, the number of blocks will be quite large. The animator will want to monitor the result of the simulation process. OpenGL is a good choice for supporting this 3D display.

Figure 18 is an example screen image from the animation paths solving program. Each small cube is one brick in the simulation. The viewing camera can be controlled by the mouse.

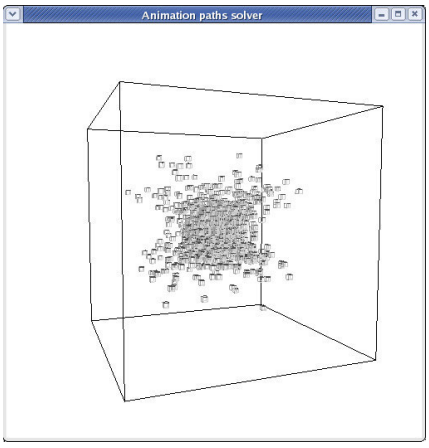

Fig. 18. One frame in the animation paths solving process.

In this program, I defined a class for the brick. The class has two arrays to store the IDs of the bricks stacked above it in the first shape and the bricks under it in the second shape. While the simulation is running, the program checks the status of each brick in the arrays to decide what this brick should do in each iteration.

IV.2. Results

Figure 19 shows several frames from the transformation between a brick model of a white cube to a brick model of a blue sphere using the flocking approach. In this case, intermediate positions were chosen from the surface of a hemisphere.

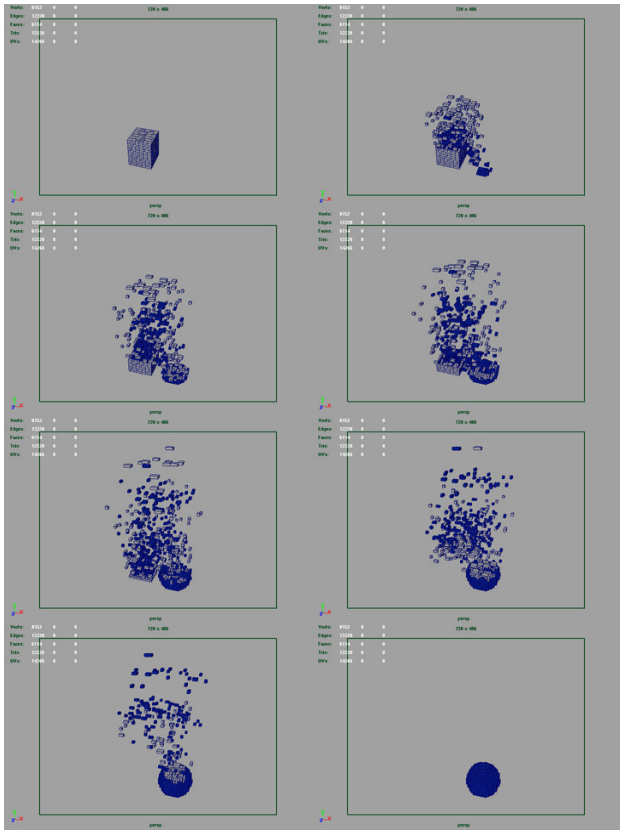

Fig. 19. A cube transforms into a sphere using the flocking approach.

Figure 20 shows another transformation from the cube to the sphere. In this case, intermediate positions were chosen from the surface of a whole sphere. The sphere is large enough to include both shapes inside. The visual effect here is very different from the first transformation example.

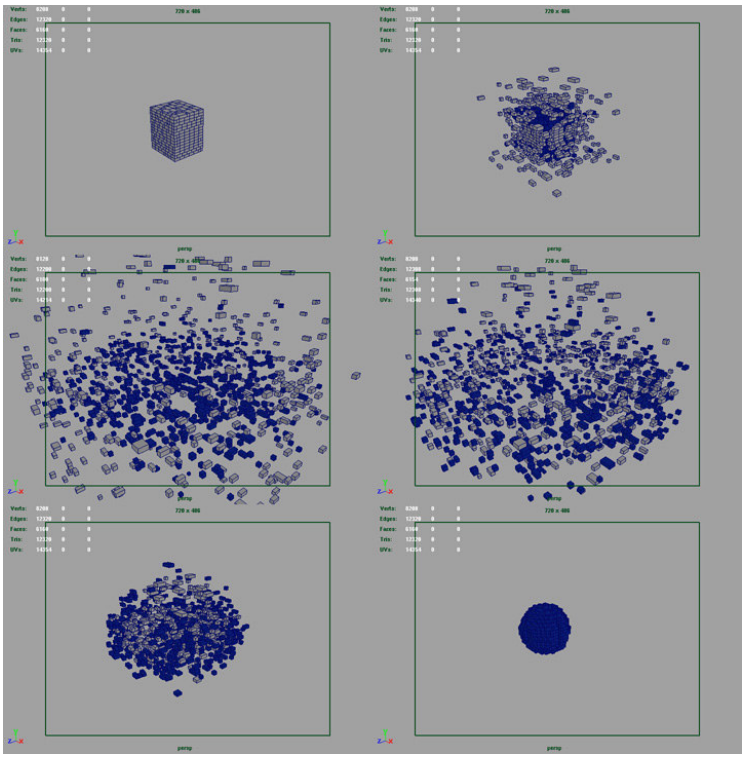

Fig. 20. A cube transforms into a sphere by using spherical intermediate positions.

The first case took five minutes to build the two shapes in MAYA. The matching program generated 988 blocks and matched them up in 15 seconds. The iterative process for determining good animation paths parameters took about 20 minutes. In each iteration the input parameters were adjusted to finally get animation paths that produced the desired result. The times for the second case are similar to the first one but took less

time in finding good parameter values.

CHAPTER V
EVALUATION, CONCLUSIONS AND FUTURE WORK

V.1. Evaluation

The original objectives have been met as the above results show. We have developed a method that automatically builds specified 3D shapes using a group of bricks with different sizes. The bricks in two shapes can be automatically matched up. By dividing each volume into two parts, different surface colors can be specified for the two shapes. After matching, several types of animation paths have been successfully procedurally generated. The animation path results were imported into MAYA successfully and used to create high quality shape transformation animations.

The animation effects produced are very interesting. In the flock simulation, each brick moves with apparent intelligence and motivation as we expected. Several distinct visual effects were created by choosing different animation path techniques. For example, by choosing intermediate positions from a sphere (Figure 9), we can imitate the crowd animation effect shown in the film *Finding Nemo* [5] (Figure 1).

A disadvantage of the flock simulation is that there is no direct control of animation time. It might take a long time for all bricks to be deactivated. The values of accelerations for the bricks, the threshold range for the bricks to be deactivated and the deceleration factors while deactivating will affect the movements and time for the bricks to stop their motion. It requires the animator to have some experience and patience to adjust the parameters to get the desired results.

The efficiency of this method is good. The entire time for an experienced animator to generate a piece of transform animation, including modeling, matching, path solving and importing into MAYA, can be less than 30 minutes.

There are some limitations in my method. The animator must try to avoid using some thin or slim shapes, like a piece of paper, or a slim stick. Those shapes might not get good result after voxeliztion. Also to divide the shape into two parts and get enough voxels inside, might require scaling them to a very big size which will make the simulation very slow.

The flocking system will generate accelerations to avoid collisions in the animation. However there might still be some collisions if some bricks have extremely high speed or several collision avoiding accelerations counteract each other. To avoid this case and keep smooth animation paths, requires a more complex flocking system.

I developed a new method to solve the matching problem. I also developed my own flocking system which has some special functions to meet special requirements. As part of this thesis work, I learned the theories about the packing problem and the flock simulation.

V.2. Future Work

We see the potential for future work improving on several aspects of this study. These include:

1. More animation effects choices should be available for the animator. We have developed several ways to compute animation paths, but obviously not all

possible ways. Every animator may have their own idea about how to control the animation. Further study might provide more interesting and exciting effects.

2. The user interface of the animation paths solving program should be more convenient for the animator to us; interactively controlling the program rather than editing a parameter file.

REFERENCES

[1] LEGO Group. LEGO. http://www.LEGO.com/

[2] Curious Pictures. Building Blocks - AT&T. Film shown at *SIGGRAPH 2002 Animation Theater*, 2002.

[3] LEGO Group. Ldraw. http://www.ldraw.org/

[4] A. Lerios, C. D. Garfinkle and M. Levoy. "Feature-based volume metamorphosis". *Proc. 22nd Annual Conference on Computer Graphics and Interactive Techniques*, page 449 – 456, Sept. 1995.

[5] Disney/PIXAR Animation Studios. Finding Nemo (film). 2003.

[6] T. Beier, S. Neely. "Feature-based image metamorphosis". *Proc. 19th Annual Conference on Computer Graphics and Interactive Techniques SIGGRAPH '92,* Volume 26 Issue 2, page 35-42, Jul. 1992.

[7] K. Miyata. "A method of generating stone wall patterns." *Proc. 17th Annual Conference on Computer Graphics and Interactive Techniques SIGGRAPH '90*, Volume. 24 Issue 4, pp. 387-394, Sept. 1990.

[8] J. Legakis, J. Dorsey and S. Gortler. "Feature-based cellular texturing for Architectural models". *Proc. 28th Annual Conference on Computer Graphics and Interactive Techniques,* pp 309-316, Aug. 2001.

[9] M. Anderson, E. McDaniel and S. Chenney. "Autonomous characters and flocking: constrained animation of flocks". *Proc. 2003 ACM SIGGRAPH/Eurographics*

Symposium on Computer Animation SCA '03, pp. 289-297, Jul. 2003.

[10] D. Thalmann, C. Hery, S. Lippman, H. Ono, S. Regelous and D. Sutton. “Crowd and group animation”. *SIGGRAPH 2004 Course Notes SIGGRAPH '04*, Article No. 34, Aug. 2004.

[11] C. W. Reynolds. “Flocks, herds and schools: a distributed behavioral model”. *Proc. 14th Annual Conference on Computer Graphics and Interactive Techniques SIGGRAPH '87,* Volume 21 Issue 4, page 25–34, Aug. 1987.

APPENDIX A
VIDEO FILES

The attached Quicktime video files are examples of transformation animations using different approaches

anim01.mov: animation without paths

anim02.mov: animation with straight paths

anim03.mov: animation by flocking simulation

anim04.mov: animation with hemispherical intermediate positions

anim05.mov: animation with spherical intermediate positions

VITA

Lu Liu

Address
401 Southwest PKWY Apt#516
College Station, Texas 77840
twols@viz.tamu.edu

Research Interests
3D Modeling and Animation Technology
Texture Painting
Developing Animation Tools Using Scripting Languages

Education

12/06	M.S. in Visualization Sciences	Texas A&M University
07/01	Bachelor of Architecture	Tsinghua University

Employment

01/06 - 07/06	PIXAR Animation Studios,CA	Technical Director Intern
02/02 – 06/03	Crystal CG Co. ltd.Beijing, China	Technical Director