INCREASING PERFORMANCE OF 3D GAMES USING PREcalcULATION

Abdullah Al Mazed  
Teaching Assistant  
gagan.mazed@gmail.com

Mirza Hasan Asif  
Student  
amirza_bd@yahoo.com

Emdad Ahmed  
Lecturer; Graduate TA  
emdadahmed@hotmail.com

Department of Computer Science and Engineering, North South University, Dhaka, Bangladesh

ABSTRACT
Computer games specially 3D games involve massive calculation before each rendering. Before each frame is rendered, lots of things like scene-graph, lighting and shadows, current frame for dynamic animation, artificial intelligence for game-characters and a lot more things need to be calculated. To achieve smoothness in game playing, the frame-rate should be as high as possible; and one way to increase FPS (Frames Per Second), is to minimize the number of calculations. Here, in this paper, we have shown how we can reduce computation time using precalculation technique. We have compared the frame-rate of two rendered scenes, one that does all calculations on the fly; another uses precalculated data-structures, algorithms, lighting maps etc. We have found that the overall performance of the second approach is better.

KEYWORDS: 3D Graphics, 3D Games, Efficient Game Engine, BSP tree, Lightmap, FPS

1. INTRODUCTION
The more calculation we will do before rendering a scene, the less the frame-rate will be. We can reduce the number of calculations by carefully identifying the calculations that we do in every cycle or in most cycles. By precalculation we mean identifying such calculations that can be replaced by some data-structures or files which we can make before the game runs. Usually making this replacement may be computationally very expensive and time complexity may be very high. But once we make this precalculation done, the game will run faster in the real time using these precalculations. In this paper, the process of creating files, data-structures etc that will be made before the game starts which will reduce the run-time computation has been called as precalculation.

Organization of the rest of the paper is as follows: In section 2 we have discussed about different types of objects in game engine, the static and dynamic objects etc. In section 3, we have discussed on rendering a scene with the help of a precalculated tree (BSP), how we can calculate it, its benefits etc. In section 4, we have discussed about reducing the calculation for lighting effects by using lightmaps. In section 5, we have analyzed the tests that we have performed to verify our ideas. The paper concludes with its recommendation for future works in section 6.

2. STATIC AND DYNAMIC OBJECTS
In a computer game we can divide the objects into two general categories:

- Static Object
- Dynamic Object

By static object we mean the object that does not change during game play. This type of object includes:

Static polygons – polygons that do not change in shape or position. For example, in a game there are buildings, tress, and roads etc. that do not change at all during the game play. The properties of the polygons that build up the objects remain constant in the whole duration.

Static light and shadows – The effects of light that need not to be changed during Game play. For example, consider a lighting effect on the table created by a table lamp. If the light or any object on the table does not move, the shadows remain fixed. We do not need to calculate shadows on each vertex time and again.

Textured animations – animations that require only change in the textures, but the Polygons of the objects need not to be changed, resized or moved. For example, we can think about a
commercial billboard. The displayed picture on that billboard changes after certain intervals, which create the animation effect. We call this type of animations static because the polygons (billboard) do not change in size or position. Dynamic Objects are those that may change their position or shape. We can subdivide Dynamic Objects into three subcategories: Dynamic polygons-polygons that may change their shape or position. For example, the polygons that make up a moving car. Dynamic lighting effect- the effect of lights that may vary from time to time. For example the lighting effect on the road created by the headlights of a moving car. Particle effect and animation- Effects that require creation or deletion of new polygons or moving or reshaping polygons. For example, we can create a smoke effect by creating a bunch of textured small polygons moving upwards.

3. INCREASING EFFICIENCY IN RENDERING STATIC POLYGONS
To render a frame in the game, we have to find out which polygons of the scenes are in the camera and need to be rendered. Since static polygons do not change their positions or shape, we can reduce the calculation for finding the polygons that need to be rendered in the current frame. One basic way is to use a tree for the static polygons. Here we have used BSP (Binary Space Partition) tree.

3.1 Precalculation: Constructing the BSP tree

Given a set of polygons in three dimensional space, we want to build a BSP tree which contains all of the polygons. The algorithm to build a BSP tree is very simple:
1. Select a partition plane.
2. Partition the set of polygons with the plane.
3. Recurse with each of the two new sets. [5] The details about BSP tree, its advantage, disadvantages have been discussed in [1].

4. INCREASING EFFICIENCY IN RENDERING LIGHTING EFFECTS
In nature, when light is emitted from a source, it is reflected off of hundreds, if not thousands or millions of objects before reaching the user's eye. Obviously, the calculations required to perfectly simulate the natural behavior of light are too time-consuming to use for real-time 3-D graphics. In 3D games, when light reflects off a surface, the light color interacts mathematically with the surface itself to create the color eventually displayed on the screen, which requires a complex computation. [15] We can reduce the computation in the run-time if we can find any technique to calculate or figure out the lighting effects. If the light-sources are static, then one very handy technique is to use lightmap.

4.1 Precalculation: Making the Lightmaps

Light mapping is one of the revolutionary Quake [11] technologies that changed the computer game industry a lot. Light mapping is also an early example of multi texture in computer games. A light map is multiplied with a texture map to produce shadows [Fig 3]. Because black has the value zero, multiplying it with any other color changes the color to black: In a light map, shadows are black and the rest is bright. Therefore, when the light map is multiplied with the texture map, shadows become black and the rest stays essentially the same:

Let, P stands for "pixel color", T for "texture color", and L for "light map color": then,
\[ P \cdot L = L \cdot r \cdot T \cdot r ; P \cdot g = L \cdot g \cdot T \cdot g ; P \cdot b = L \cdot b \cdot T \cdot b ; \]
The detailed procedure and algorithms for generating the lightmaps can be found in [8]

5. EXPERIMENT ON PERFORMANCE EVALUATION

5.1 Configuration of the PC
All the experiments have been conducted using the PC with Intel Pentium 4 HyperThread, 2.8 GHz processor, 512 MB DDR RAM, Microsoft Windows XP SP2 (Hyper Thread enabled). The video card we have used is Intel Extreme Graphics (Onboard) with 32 MB Video Memory.

5.2 Experiment on BSP tree
In this experiment we want to compare the performance of rendering a world with BSP tree and without BSP tree. We have performed 3 tests for comparison.

<table>
<thead>
<tr>
<th>Experiment Name</th>
<th>Purposes</th>
</tr>
</thead>
<tbody>
<tr>
<td>BSP tree construction time</td>
<td>This will show the time complexity of BSP tree construction (Precalculation)</td>
</tr>
<tr>
<td>Game loading time</td>
<td>This will show whether loading a world with BSP tree is faster or not.</td>
</tr>
<tr>
<td>FPS</td>
<td>This is the main test. We want to test whether the performance in rendering a world increases with the usage of BSP tree.</td>
</tr>
</tbody>
</table>

Table 1: Experiment name and its Purposes
In this experiment we have executed the tests against increasing number of polygons (from 50,000 to 3,00,000), the results of which can be found in Figure 6 to 9.

5.2.2 Experimental setup for testing BSP tree
5.2.2.1 Creating the world

First we have created a world in Discreet 3D Studio Max 6.0 [10]. We have 15 identical boxes (14 of them were created as reference clone of same box) and one more box under the camera. The boxes are arranged in such a way that they are scattered all around the camera and none of them intersect each other. [Fig. 4]

To create a world with increased number of polygons, we have used the same world, but we have just changed the parameters of the boxes. We have increased the number of segments of the boxes (in 3DS MAX these parameters are: Length segs, Width segs, height segs.) to increase the number of polygons. The “Polygon counter” tool has been used to count the number of polygons.

5.2.2.2 Converting the worlds for game and writing codes to render
We have used 3DSTATE version 6 (C++, student license) [9] as our game engine. We have used the converter provided by the engine to convert the world from 3DS MAX to Engine’s own format (.wld extension). To render this world we have written simple C++ codes using 3DSTATE API.

5.2.3 Collecting experimental data
5.2.3.1 Collecting data for BSP construction time

When we render the world using the game engine for the first time using USER_DEFINED_BEHAVIOR mode, the BSP tree is built by the engine and a file named as “worldName.bsp2” is created. We look in the log file [Fig. 5] generated by the engine to get the construction time.

5.2.3.2 Collecting data for game loading time
We have run the game several times and calculated the average time taken to load the game by the engine. We have excluded the very first run from this calculation because that constructs the BSP tree. The mode of game engine is set to
USER_DEFINED_BEHAVIOR to load and render the world using BSP tree and to EDITOR_MODE to load and render without using BSP tree. We have taken the time before and after the function STATE_engine_load(...) and thus calculated the loading time.

5.2.3.3 Collecting data on FPS

```c
while()
{
  //game loop
}
int time = (int)STATE_engine_get_average_render_execution_time();
fprintf(file,"FPS = %.2lf\n",(double)1000000.0/(double)time);
//converting to FPS
```

The BSPTree was built with the following parameters:
- BSP precision = 1.000000
- Division Coefficient = 2.000000
- Split coefficient = 80.000000
- Good enough score = 80.000000

![Figure 5: Snippet from the log file generated by the game engine](image)

To calculate the FPS, after the game loop (while loop) we have taken the average render time and using that we calculated the FPS.

5.3 Results and analysis

5.3.1 BSPTree construction time

The graph clearly shows that the complexity increases with the increasing number of polygons. This result indicates that we do not need to worry much on the time complexity of BSPTree construction. If the time complexity to build BSPTree had been exponential, then we would have to reconsider building BSPTree for the game that has huge number of polygons. Fortunately for us, we see from the graph that it is NOT exponential.

![Figure 6: BSPTree construction time](image)

Table 2: BSPTree Construction time against number of polygons

<table>
<thead>
<tr>
<th>Polygons</th>
<th>50,413</th>
<th>1,00,813</th>
</tr>
</thead>
<tbody>
<tr>
<td>BSP construction time (minute)</td>
<td>0.013550</td>
<td>0.02682</td>
</tr>
<tr>
<td>1,51,392</td>
<td>0.04533</td>
<td>0.09005</td>
</tr>
<tr>
<td>2,03,143</td>
<td>0.06618</td>
<td>0.07147</td>
</tr>
<tr>
<td>2,54,413</td>
<td>0.09912</td>
<td></td>
</tr>
</tbody>
</table>

5.3.2 Game Loading time

![Figure 7: Game Loading Time](image)
Table 3: Game loading time against number of polygons

<table>
<thead>
<tr>
<th>Number of Polygons</th>
<th>50,413</th>
<th>1.00,813</th>
<th>1.51,393</th>
<th>2.03,413</th>
<th>2.54,413</th>
<th>3.00,912</th>
</tr>
</thead>
<tbody>
<tr>
<td>Game Loading Time (Sec)</td>
<td>5.562</td>
<td>11.235</td>
<td>18.579</td>
<td>26.203</td>
<td>45.141</td>
<td>66.704</td>
</tr>
<tr>
<td>With BSP tree</td>
<td>5.422</td>
<td>11.820</td>
<td>19.683</td>
<td>27.469</td>
<td>45.785</td>
<td>68.312</td>
</tr>
<tr>
<td>Without BSP tree</td>
<td>5.422</td>
<td>11.820</td>
<td>19.683</td>
<td>27.469</td>
<td>45.785</td>
<td>68.312</td>
</tr>
</tbody>
</table>

Fig. 7 shows an interesting fact that firstly (when number of polygons is comparatively small, say upto 1,50,000) game loading time is higher with BSP tree. But as we increase the number of polygons (above 1,50,000), the game with BSP tree loads faster. So it suggests we better use BSP tree even if we just want the game to be loaded faster.

5.3.3 FPS (Frames Per Second)

Figure 8: FPS (Frames Per Second)

<table>
<thead>
<tr>
<th>Number of Polygons</th>
<th>50,413</th>
<th>1.00,813</th>
<th>1.51,393</th>
<th>2.03,413</th>
<th>2.54,413</th>
<th>3.00,912</th>
</tr>
</thead>
<tbody>
<tr>
<td>FPS with BSP</td>
<td>85</td>
<td>83.22</td>
<td>76.27</td>
<td>65.84</td>
<td>63.69</td>
<td>44.54</td>
</tr>
<tr>
<td>FPS without BSP</td>
<td>45.27</td>
<td>25.15</td>
<td>14.91</td>
<td>11.63</td>
<td>9.6</td>
<td>7.5</td>
</tr>
</tbody>
</table>

Our main idea was to prove that if we use a precalculated BSP tree, the performance of the 3D game will increase. Fig 8 clearly shows that the game with BSP tree has always higher FPS than the one without BSP tree, and we know that the higher the FPS, the smoother the game.

5.4 Experiment on Lighting Effect
5.4.1 Creating the world

Here we have used the same worlds that were used in the previous experiments. We just added four omni light around the camera. To create the world with lightmap, we used render to texture in 3DS Max and used the tool provided by the game engine to convert to the engine format.

5.4.2 Collecting data on FPS

The process is the same as before. We rendered the two different types of worlds that we have created, one with lightmap, another without lightmap. Then we have collected the FPS data as we have done before.

5.4.3 Results and analysis

Figure 9: FPS with lighting effect

Table 5: FPS with lighting effects

<table>
<thead>
<tr>
<th>Number of Polygons</th>
<th>50,413</th>
<th>1.00,813</th>
<th>1.51,393</th>
<th>2.03,413</th>
<th>2.54,413</th>
<th>3.00,912</th>
</tr>
</thead>
<tbody>
<tr>
<td>FPS without lightmap</td>
<td>84</td>
<td>77</td>
<td>69.23</td>
<td>55.78</td>
<td>48.92</td>
<td>22.87</td>
</tr>
<tr>
<td>FPS with lightmap</td>
<td>85</td>
<td>83.11</td>
<td>76.02</td>
<td>64.94</td>
<td>60.22</td>
<td>42.69</td>
</tr>
</tbody>
</table>

The result of the test is obvious from the figure. The world with lightmap is rendered with higher FPS. In both cases FPS decreases with the increase of the number of polygon. The result is consistent with our theories. Since without lightmap, all the lighting effects need to be calculated runtime, it is expected that it will take more time, and therefore result in less FPS.
6. OPEN RESEARCH ISSUES
Here is a list of open research issues relevant to precalculation that increases the performance of 3D game:

- Precalculation of a better tree or structure of rendering static polygons.
- Development of faster algorithms for generating lightmaps.
- Researching on integration of precalculated trees for rendering dynamic polygons.
- Researching on intelligent techniques for combining lightmaps to simulate dynamic lighting effects.

7. CONCLUSION
The main objective of using precalculation is to reduce computation during runtime, even at a high cost of precalculation. But since this precalculation is done only once, and before the final release of the 3D game, we can sacrifice that. The experiments that we have performed also verify our claims. Any 3D game engine can utilize the precalculation techniques we discussed to improve performance and flexibility.

REFERENCES
15. DirectX Documentation for C++, DirectX 9.0c.