Design and Performance Evaluation of Soft 3D Models using Metaball in Unreal Engine 4

Youngsik Kim

Dept. of Game and Multimedia Engineering, Tech University of Korea, Republic of Korea
E-mail: kys@tukorea.ac.kr

Received 27 April 2021; Accepted 06 January 2022;
Publication 05 July 2022

Abstract

Soft 3D models such as Slime need to use a variety of different animations in the same situation in order to show a natural dynamic appearance. Metaball can express soft objects well with a small amount of data, but it requires a lot of computation time for real-time rendering. Because of this, it is difficult to find models using Metaball in 3D games. This paper developed a 3D slime game character using Metaball by applying ray marching technique in Unreal Engine 4. In addition, 3D games including slime characters and general fixed game character models were produced and the performance of it were evaluated. Even if the number of slime characters is changed from 0 to 40, it has been verified that the rendering speed is maintained at 30 FPS (Frames Per Second) or more, so that the game can be played.

Keywords: Soft 3D models, Slime, Unreal Engine 4, Metaball, performance evaluation.

Journal of Mobile Multimedia, Vol. 18_6, 1795–1810.
doi: 10.13052/jmm1550-4646.18614

1 Introduction

In modern 3D games, modelling and production of fixed game characters includes many animations, but the same animation is played in the same situation. However, soft game character models, such as fluid slime, must use a variety of different animations in the same situation in order to show a natural dynamic appearance. However, in many cases, one animation is processed in the same situation due to various animation production costs and performance problems of indeterminate game characters.

There has been a lot of research on using Metaball when creating game enemies. Metaball belongs to the implicit surface model. The modelling elements formed as point elements have been called differently according to the field function representing the distribution of density values in the space for determining the surface. Blinn is called Blobs, Nishimura is Metaball, and Wyvill is a soft object [1]. With the above techniques, modelling using arbitrary skeletal elements such as curves and polygons has been widely applied both academically and commercially [2].

Metaball can represent objects well with a small amount of data. When expressing a model of a soft object, there is no need to create a separate animation through real-time calculation, and it is possible to show various dynamic shapes according to the situation [3].

Some computer graphics modelling programs use Metaball to model characters. Auto Desk's 3Ds Max also uses a technique called metaball modelling, which is suitable for simulation of liquid and thick viscous substances such as mud, soft food, and dissolved metal. Figures 1 and 2 are a part of the process of making a simple bug, a metaball model, using the blob mesh composite object of 3ds Max. This is how the sphere is placed in Figure 1 and the result is created as shown in Figure 2. In the past, there have been attempts to use it as a modelling technique.

Despite these advantages, there are problems that must be overcome in order to apply the Metaball technique to a negative 3D game character. First, the real-time rendering of 3D game characters using the Metaball technique requires a large amount of computation, so high hardware performance or optimized algorithms are needed [4]. However, in recent years, computer graphics hardware performance has improved dramatically, and it has become possible to solve to some extent due to improved algorithms.

The second problem is that it can produce unwanted results. Due to the principle of Metaball, it cannot prevent the nature of sticking to each other between adjacent elements. So, if you limit the area to be attached, you may

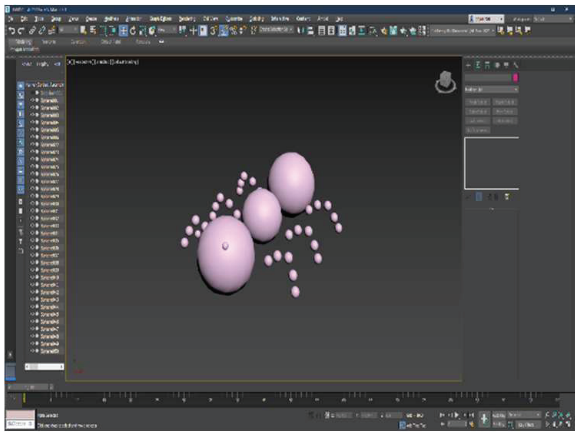


Figure 1 Sphere arrangement for BlobWorm.

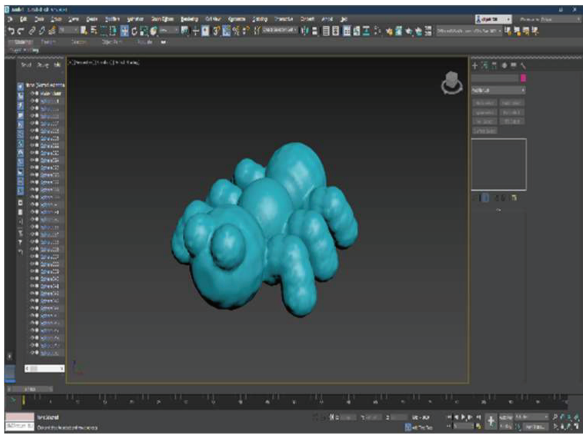


Figure 2 Metaball model of BlobWorm.

get an awkward model or you may have trouble getting a smooth surface. The second drawback is that there is no problem with the model of a soft object such as slime, which is not a complex modelling, and the first drawback is that the burden of real-time rendering has been greatly reduced due to the advancement of technology such as algorithms and hardware.

This paper, using a custom node in the Unreal Engine 4 material, implements Metaball using ray marching as a shader to create an indeterminate model such as slime. In 3D games, this paper will measure whether the performance enough to be used in real games is achieved by maintaining FPS while performing calculations that move like regular fixed game character models and receive the laws of physics.

2 Metaball Algorithm

Since the concept of Metaball (blobs or soft objects) was introduced in the early 80s, many related studies have been conducted because of its excellent expressive ability [3]. However, since real-time rendering was difficult due to a high amount of computation, various algorithms have been developed and improved. A new field function and fast ray tracing algorithm [4], which are good for representing objects that require very natural connections such as fluid shape expressions by proposing a field function of the 6th order polynomial, traces rays through inertial interpolation to the existing ray tracing technique. A number of methods have been studied, such as IRCF (Improved Ray Casting Function) [5], which is a technique to perform, and acceleration of volume ray projection using a modified marching cube table [6].

There are two methods of expressing complex-shaped surfaces: parametric surface expression and implicit surface model. Implicit surface models are commonly referred to as isosurface models. The field value at a point in space is determined by the ratio of the distance from the center of the metaball to the radius of the metaball, and a distribution body having a field function whose field value at the surface of the sphere becomes 0 is called a metaball [4].

The isosurface means a surface with a constant electron density. The recently used density function uses a more simplified form of Wyvill's calculation formula [9]. Since a curved surface made of metaballs exhibits algebraic characteristics, it is relatively simple to implement in a system that supports raytracing, but a system that expresses all objects as a combination of polygons such as 3DS MAX uses an algorithm called Marching Cubes.

The Marching Cubes algorithm is a method of dividing the space affected by ellipsoids into small voxels, and then calculating the density function at the vertices of each hexahedron to find the plane that exists inside the cube. Since there are 8 vertices of the hexahedron and two cases are possible inside and outside the isosurface, a total of 256 can occur. Among them, 14 cases remain, except for geometrically identical ones that can be rotated.

Metaball can be created by storing the numbers in 14 cases in advance in an array of code and generating the right shape for each voxel according to the density value. It can also be implemented in Unreal Engine, but as the size increases, the computational amount increases and the number of voxels is increased to create a smooth surface.

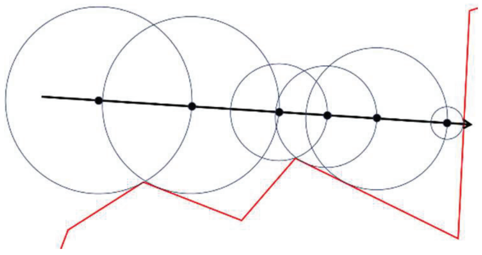


Figure 3 Ray marching.

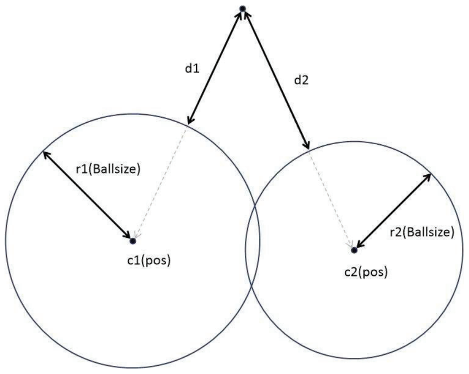
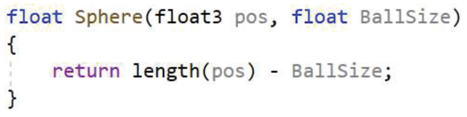


Figure 4 SDF sphere.



```
float Sphere(float3 pos, float BallSize)
{
    return length(pos) - BallSize;
}
```

Figure 5 Source code of SDF sphere.

This paper will create a blob with a field function using Ray Marching and implement it as a calculation that expresses a curved surface. Ray Marching literally proceeds as the ray advances. Each fragment fires a ray, advancing the ray by the minimum distance of the map calculated from the field function. This paper will find the minimum distance using one of Ray Marching's algorithms, Sphere Tracing. As shown in Figure 3, find the largest sphere that can be touched at the current ray position, and if it touches in the ray direction, stop the process and move on to the next fragment. If not, move forward by the minimum distance and repeat the same process again. Here, if the ray is not blocked, it repeats infinitely, so you must set the maximum

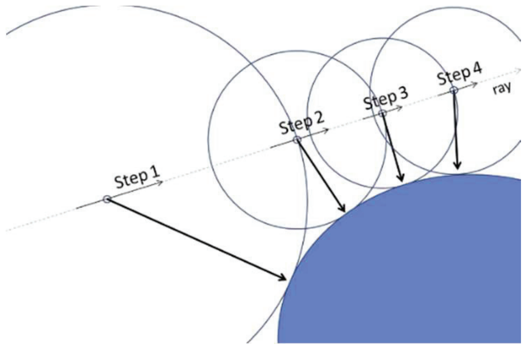


Figure 6 SDF action in ray marching.

number of iterations, MaxSteps, to limit it. This number of iterations affects the performance and the precision of the metaball output.

To use this ray marching, you need to find the minimum distance from the current ray position. To do that, this paper will use the Signed Distance Function (SDF) [9]. Simply, if the point is outside the object, it displays the shortest distance to the object, if it is above the object boundary, it displays 0, and if it is inside, it displays the shortest distance x or -1 to the boundary line. As a simple example, to show the source code of Figure 5 for drawing a circle, as shown in Figure 4, you can obtain the length of pos that will be the center point of the sphere (c1, c2) and subtract it by the radius (r1, r2) of the ball size to obtain the minimum distance (d1, d2). Then, for each fragment, the closest distance that can be seen from the circle can be obtained, so that a sphere can be drawn. Connect the spheres created in this way with a smoothing function, find and return the minimum distance among all SDF values, and advance the ray by the value.

As shown in Figure 4, the minimum distance (d1, d2) can be obtained by subtracting the length of pos that will be the center point (c1, c2) of the sphere and subtracting it by the radius (r1, r2) of the ball size. Then, for each fragment, the closest distance that can be seen from the circle can be obtained, so that a sphere can be drawn. Connect the spheres created in this way with a smoothing function, find and return the minimum distance among all SDF values, and advance the ray by the value. Figure 6 shows the state of advancing with the above SDF value. As metaball technology advances, functions are connected to smooth surfaces and perform efficient calculations.

The Blinn's field function of Equation (1) proposed in 1982 [7] has a disadvantage in that it is an exponential function that requires all metaballs to be considered because even the field value at a very far distance from the center point is not zero.

$$f_i(r) = e^{-ar^3} \tag{1}$$

The Nishimura's field function of Equation (2) proposed in 1983 [8] has a relatively small computational time, but has a large error and a large number of sections to be solved when several metaballs are overlapped.

$$f_i(r) = \begin{cases} 1 - 3\left(\frac{r}{R_i}\right)^2 & \text{if } \left(0 \le r < \frac{R}{3}\right) \\ \frac{3}{2}\left(1 - \left(\frac{r}{R_i}\right)\right)^2 & \text{if } \left(\frac{R}{3} \le r \le R_i\right) \end{cases} \tag{2}$$

A display function that uses the field function of Equation (3) of a polynomial that approximates the exponential function of Wyvill developed in 1986 [9] was developed and called it a soft object.

$$f_i(r) = -\frac{4}{9}\left(\frac{r}{R_i}\right)^6 + \frac{17}{9}\left(\frac{r}{R_i}\right)^4 - \frac{22}{9}\left(\frac{r}{R_i}\right)^2 + 1 \tag{3}$$

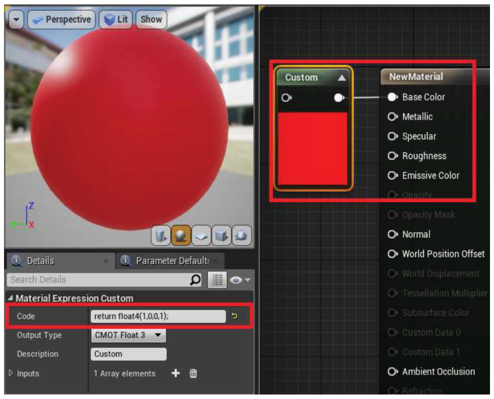
Metaball's connection to the field function above was unnatural. So, for smooth connection, Eun-seok Kim proposed a new field function in Equation (4) [5]. This resulted in more efficient results in the average number of sections examined to find roots for one ray than the previous functions [5]. Using the above methods, this paper can create metaballs, connect them, and render them.

$$f_i(r) = \left(1 - \left(\frac{r}{R_i}\right)^2\right)^3 \tag{4}$$

3 Design of Metaball in Unreal Engine 4

In Unreal Engine 4, custom shaders can be created using High Level Shading Language (HLSL). Create a custom node after creating a material as shown in Figure 7. After that, you can write shader code like hlsl in the code and apply it according to node connection. Here, you can set the material more easily by adjusting the values of Metallic, Specular, and Roughness. This paper will apply the Metaball algorithm to this line of code.

To use the metaball algorithm, you need to be able to store and change the location of blobs. This paper will save it using the vector parameter in



Material Expression Custom

Code	return float4(1,0,0,1);
Output Type	CMOT Float 3
Description	Custom
Inputs	1 Array elements

Figure 7 Custom shader in Unreal Engine 4.

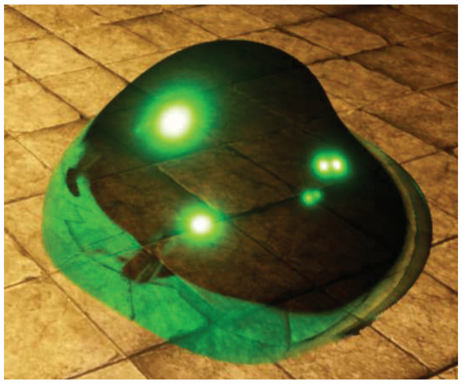


Figure 8 Slime of ShieldMan game project in Unreal Engine 4.

the material and change the position of the blob with the SetVectorParameterValueOnMaterials function in C++. Describes the process of creating a Metaball model in Unreal Engine 4. On top of the metaball, the settings from Unreal Materials are added to create a texture. Figure 8 shows ShieldMan's Slime, a 3D game project created using Unreal Engine version 4.23. The material created above is covered with a StaticMesh that is large enough for the metaball to be drawn. Then, the Custom Shader is rendered only as much as the size of the StaticMesh. Now this paper need to move and physically move this slime and make the blobs move accordingly. This part was implemented using Unreal's physics system and c++.

Create a new c++ class by inheriting the Actor class. Create a StaticMesh variable with the Metaball material above, and blobs in the Metaball shader

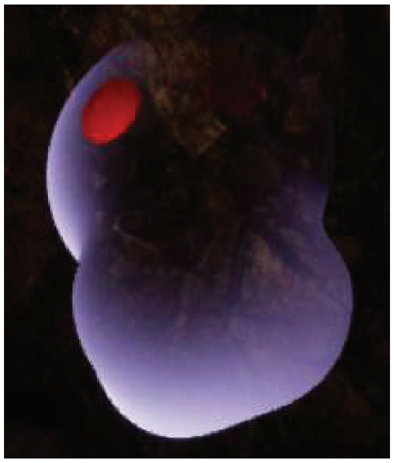


Figure 9 Slime ghost of ShieldMan game project.

cannot be applied to physics with Unreal Engine, so you have to put them in code. In ShieldMan project, Spring Mass System was used to make natural movement. Because the principle is simple and can be easily implemented, it is very easy to link with other physical models. Since it shows a spring-like motion with an additional mass, it can also express the dynamic motion of complex natural phenomena linked to deformation, destruction, fire, water, and explosion [10]. Each sphere's Position and Velocity are stored and used in an array using Unreal's FVector. As the position is updated one by one while rotating the for statement, performance decreases as the number of spheres increases.

Slime's movement was adjusted using AddForceVelocity to give it a bouncing feel, and a NavigationSystem was used to show the roaming appearance, allowing him to continue to move to a random position within the Navi mesh bound volume installed on the editor. Metaball is also a good option for ghosts of uncertain shape. The slime in Figure 8 was modified to create the ghost in Figure 9. This paper changed the alpha value and made the eye part rather wide and changed the color. You can create multiple models using Metaball using these various application methods.

4 Performance Evaluation

For performance evaluation, this paper will check whether there is any problem in the application of the actual game by measuring the GPU usage

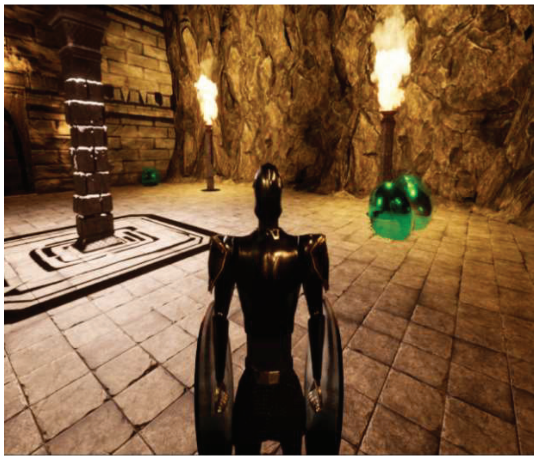


Figure 10 Screen shot of ShieldMan game project in unreal engine 4.

or FPS of the ShieldMan project that is playing the game by putting Metaball models. Figure 10 is a screen shot of the ShieldMan game project.

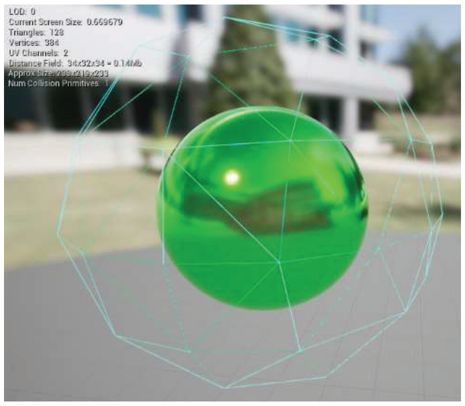
4.1 Performance Evaluation Environment

Performance evaluation was measured on a CPU: Intel(R) Core(TM) i7-7700HQ CPU @ 2.8GHz, memory: 8.0GB, GPU: NVIDIA GeForce GTX 1050Ti computer. The Unreal Project environment will use the levels shown in Figure 11. Character BP using a total of 73 StaticMesh, 10 Particle System, 11 Point Light, and 1 Physics Asset is composed.

Slime will use Metaball with 6 spheres, and as the material setting, use Translucent as the Blend Mode, set the Screen Space Reflections to true, and set the Lighting Mode to Surface ForwardShading, so the influence of each light is calculated per pixel, so the most expensive translucent lighting method is used. Ray Marching's MaxSteps is 200, and the size of the StaticMesh to which the shader is rendered is 233 $\times$ 219 $\times$ 233 in Unreal Engine 4, and has 128 Triangles and 384 Vertices. The size of StaticMesh and Slime are roughly the same.

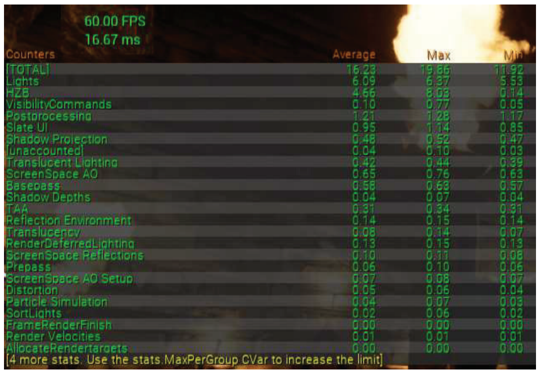
4.2 Performance Evaluation According to the Number of Slimes

This paper will compare the number of Slime by measuring the FPS at 0, 5, 10, 20, 40. The location was measured from the corner where all maps can be seen. However, when slime moves and occupies a large part of the screen,



```
LOD: 0
Current Screen Size: 0.669679
Triangles: 128
Vertices: 384
UV Channels: 2
Distance Field: 34x32x34 = 0.14Mb
Approx Size: 233x219x233
Num Collision Primitives: 1
```

Figure 11 StaticMesh information.



```
         60.00 FPS
         16.67 ms
Counters                              Average      Max      Min
[TOTAL]                                 16.23    19.86    11.92
Lights                                   6.09     6.37     5.53
HZB                                      4.66     8.03     0.14
VisibilityCommands                       0.10     0.77     0.05
Postprocessing                           1.21     1.28     1.17
Slate UI                                 0.95     1.14     0.85
Shadow Projection                        0.48     0.52     0.47
[unaccounted]                            0.04     0.10     0.03
Translucent Lighting                     0.42     0.44     0.39
ScreenSpace AO                           0.65     0.76     0.63
Basepass                                 0.58     0.63     0.57
Shadow Depths                            0.04     0.07     0.04
TAA                                      0.31     0.34     0.31
Reflection Environment                   0.14     0.15     0.14
Translucency                             0.08     0.14     0.07
RenderDeferredLighting                   0.13     0.15     0.13
ScreenSpace Reflections                  0.10     0.11     0.08
Prepass                                  0.06     0.10     0.06
ScreenSpace AO Setup                     0.07     0.08     0.07
Distortion                               0.05     0.06     0.04
Particle Simulation                      0.04     0.07     0.03
SortLights                               0.02     0.06     0.02
FrameRenderFinish                        0.00     0.00     0.00
Render Velocities                        0.01     0.01     0.01
AllocateRendertargets                    0.00     0.00     0.00
[4 more stats. Use the stats.MaxPerGroup CVar to increase the limit]
```

Figure 12 Performance profiling of 0 slime.

there is a difference in FPS dropping. As shown in Figure 12, it was measured using the commands stat GPU and stat FPS in Unreal Engine 4. The stat FPS command displays the number of frames currently being rendered per second and the time it took to render the frame in milliseconds (ms), and the stat GPU's GPU time measures the time it takes the video card to render the scene. The GPU time is synchronized to the frame, so it is likely to be similar to the frame time. The number was increased from when there was no slime to 40 and recorded in the table. Initially, as shown in Figure 12, it takes a lot of calculation cost for lights calculation in GPU, and the more the number of slimes increases, the most calculation cost of translucency becomes. When it is 0, 60 fps is maintained and it can be confirmed that the play is possible without any problems.

Table 1 Performance according to the number of slimes

Number of Slimes	GPU Rendering Time (ms)	Rendering Speed (FPS: Frames Per Second)	Rendering Time (ms)
0	16.23	60	16.67
5	16.14	59.98	16.67
10	20.75	45.43	22.01
20	22.69	41.99	23.81
40	30.52	31.91	31.34

From the results of Table 1, it can be seen that the 5 animals have little effect on the game play. In the planning of the ShieldMan project, the goal was to catch about 5 Slimes, so there is no problem in creating and using models with Metaball technology. In contrast, 10 cases showed a big drop. It may be a difference depending on the movement or position of the slime, but even considering such points, it is a number that allows the player to recognize that the frame is falling. Nevertheless, up to 20 or 40 frames may appear broken, but play is possible. If you run the optimization to the maximum, you will be able to sufficiently defend the frame.

However, considering the Metaball technology and Unreal Engine 4's material system, the more StaticMesh using the Metaball material is visible on the screen, the more fragments the Ray Marching proceeds. And the more the empty parts are, the more the number of advances increases, so the percentage of the viewport is important. Therefore, the percentage of the viewport should be considered as a measure of performance.

4.3 Performance Evaluation According to Viewport Occupancy Ratio

The environment for this performance evaluation will use a large metaball model with 27 spheres, arranged in $3 \times 3 \times 3$ like a cube, and a pattern that rotates like when one side fits a cube. As the material setting, using Translucent as the Blend Mode, the Screen Space Reflections as true, and the Surface TranslucencyVolume as the Lighting Mode, the lighting for the surface is calculated, the cost per pixel is very low, and only diffuse lighting is supported. The size of the StaticMesh is $700 \times 700 \times 700$ in Unreal notation and has 12 Triangles, 24 Vertices.

The Unreal Project environment will use the levels shown in Figure 13. Character BP using a total of 89 StaticMesh, 12 light emitting lava material, 2 Particle System, 14 Point Light, 2 SpotLight, and 1 Physics Asset are

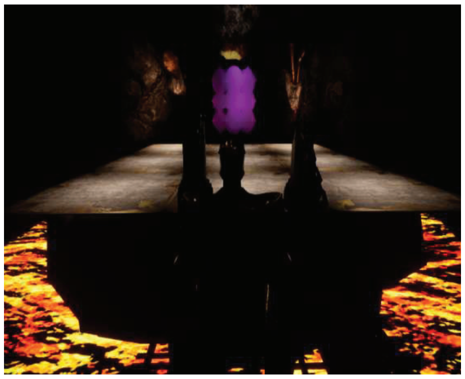
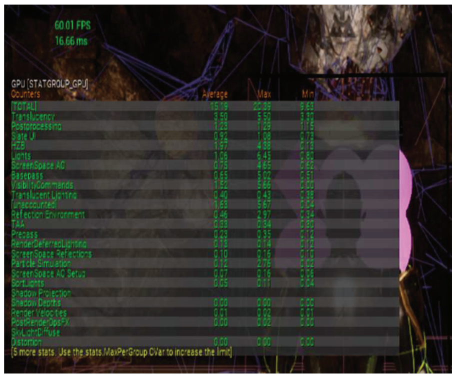


Figure 13 Screen shot of ShiedMan game project in cube level.



```
60.01 FPS
16.66 ms

GPU [STATGROUP_GPU]
Counters                      Average   Max     Min
[TOTAL]                       15.19     20.39   9.63
Translucency                  3.50      5.50    3.32
Postprocessing                1.23      1.29    1.16
Slate UI                      0.92      1.08    0.73
HZB                           1.97      4.38    0.13
Lights                        1.06      6.45    0.80
ScreenSpace AO                0.75      4.65    0.52
Basepass                      0.65      5.02    0.51
VisibilityCommands            1.52      5.66    0.00
Translucent Lighting          0.40      0.43    0.38
[unaccounted]                 1.63      5.67    0.04
Reflection Environment        0.46      2.97    0.34
TAA                           0.33      0.34    0.30
Prepass                       0.23      0.35    0.12
RenderDeferredLighting        0.13      0.14    0.12
ScreenSpace Reflections       0.10      0.16    0.10
Particle Simulation           0.12      2.76    0.02
ScreenSpace AO Setup          0.07      0.16    0.06
SortLights                    0.05      0.11    0.04
Shadow Projection
Shadow Depths                 0.00      0.00    0.00
Render Velocities             0.01      0.02    0.01
PostRenderOpsFX               0.00      0.02    0.00
SkyLightDiffuse
Distortion                    0.00      0.00    0.00
[5 more stats. Use the stats.MaxPerGroup CVar to increase the limit]
```

Figure 14 Performance profiling of 400 × 400 size, 11% of the screen.

Table 2 Performance according to the occupied screen size

Occupied Screen Ratio (%)	GPU Rendering Time (ms)	Rendering Speed (FPS: Frames Per Second)	Rendering Time (ms)
0%	10.85	60.01	16.66
11%	15.19	60.01	16.66
30%	18.96	51.68	19.35
66%	31.55	30.57	32.71
100%	41.41	23.55	42.46

composed. The performance evaluation was measured according to the area of the metaball shown on the screen, and then changed the MaxSteps when the metaball was full on the screen. Compared to StaticMesh, the area occupied by the metaball is approximately 40%, and for 60% of the blank space, the for statement runs until MaxSteps, so the difference in performance varies greatly depending on MaxSteps. When making area measurements, this paper sets MaxSteps to 100 and move the character's position to change the percentage of the screen. In order to assume the play situation, the Metaball model applies the Spring Mass System as it is, and proceeds with the pattern of matching cubes while turning one side. At the screen resolution of 1600×900, the approximate area of the part occupied by the StaticMesh was calculated and expressed as a rounded percentage.

As shown in the results of Table 2, the percentage occupying about 10% on the screen did not affect the FPS. However, even if it is only 30%, it can be seen that it affects a large proportion, and if it exceeds 60%, it is about 30 frames, which makes it difficult to play. Figure 14 shows the performance profiling of 400×400 size when Occupied Screen Ratio is 11%. When MaxSteps is 100, if the space of Metaball Shader occupies a large part of the screen, it can be seen that gameplay is disrupted. So, when using Metaball, it will be important to ensure that it occupies only the necessary part. This time, while keeping the percentage of the screen at 100%, this paper measured while changing MaxSteps.

5 Conclusion

Metaball technology is advantageous for expressing fluid motion. Since various forms can be expressed with a small amount of data, if character modeling is performed using these points, it is possible to show various appearances without creating animations individually. This paper implemented Metaball using Ray Marching in Unreal to measure its performance. After creating a model by implementing Metaball using Ray Marching in a custom node of the Unreal Material System, this paper confirmed that it is enough to use AI to play the actual game by creating an actor in c++.

The five slimes didn't affect the FPS, and even when using a huge and complex model, you could keep the frame rate by adjusting the percentage of the screen and MaxSteps. In the ShieldMan project, 3D game characters Slime and Ghost that actually use Metaball models are created, and the Boss model is added to play. As a result of the performance verification, it is not unreasonable to use more complex models "without" more optimized algorithms. In order to apply the Metaball model to the game, it will be able to maintain FPS and use good graphics by adjusting the appropriate size and MaxSteps with an optimized algorithm.

Acknowledgement

This work was supported by Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIP) (No. 2016-0-00204, Development of mobile GPU hardware for photo-realistic real time virtual reality).

This work was supported by the 2022 sabbatical year research grant of the Tech University of Korea.

References

[1] Eun Seok Kim, Jay Jeong Kim, "MetaCube : A New Skeletal Element for Modeling Informal Objects" , Journal of KIISE : Computer Systems and Theory 27(4), 353–361 (9 pages), 2000.4.

[2] M.K. Park, E.T. Lee, "Modeling with Implicit Surface", ETRI Electronics and Telecommunications Trends, Vol. 13, No. 3, 53–60 (7 pages), 1998.6.

[3] Jay Jeong Kim, "A Study on an Automatic Description of Volumetric Objects using Metaballs", Korea Science and Engineering Foundation, 961-9010-057-1, 1997.2.

[4] En Suk Kim, Jay Jeong Kim, "A new field function for improvement of modeling capability and a fast ray tracing algorithm in metaball modeling", The Korean Institute of Information Scientists and Engineers collection of academic papers, 23(1A), 739–742 (4 pages), 1996.4.

[5] Jin-Youl Kim, Hyeong-Gyun Kim, "Volume Rendering by Improved Ray Casting" , Korea Multimedia Society collection of academic papers, 464–467 (4 pages), 2003.5.

[6] Sukhyun Lim, Ju-Hwan Kim, Byeong-Seok Shin, "Volume Ray Casting Acceleration Using Modified Marching Cubes Tables" , The Korean Institute of Information Scientists and Engineers collection of academic papers, 35(2A), 217–218 (2 pages), 2008.10.

[7] Blinn, J. F., "A Generalization of Algebraic Surface Drawing," ACM Transactions on Graphics, Vol. 1, No. 3, pp. 235–256 (21 pages), July 1982.

[8] Nishimura, H., Hirai, M., Kawai, T., Kawata, T., Shirakawa, I. and Omura, K., "Object Modeling by Distribution Function and a Method of Image Generation," Trans. IEICE Japan, Vol. J68-D, No. 4, pp. 718–725 (7 pages), 1986.

[9] Wyvill, G., MacPheeters, G. and Wyvill, B., “Data Structure for Soft Objects,” SIGGRAPH ’93 Course Notes, Vol. 25, pp. 227–234 (7 pages), August 1993.
[10] Bong-Jun Kim, Jeong-Mo Hong , “Interaction fluid and deformation model based mass-spring” , Korea Computer Graphics Society academic conference, 65–66 (2 pages), 2014.7
[11] Unreal Engine 4 in http://www.unrealengine.com/

Biography

Youngsik Kim received the B.S., M.S., and Ph.D degree in Dept. Computer Science from the Yonsei University, Korea, in 1993, 1995, and 1999 respectively. He had worked for System LSI, Samsung Electronics Co. Ltd from Aug. 1999 to Feb. 2005 as a senior engineer. Since March 2005 he has been working for Dept. of Game & Multimedia Engineering in Tech University of Korea. His research interests are in 3D Graphics and Multimedia Architectures, Game Programming, and SOC designs.