Real-time river simulation using metaballs

Anders Steen
astee@kth.se

May 11, 2020

Abstract

Simulating liquids in real time is a difficult problem. Using metaballs to simulate water in real time is an approach that has been used before in different ways. This paper aims to explore the advantages and drawbacks of river simulation in real time using metaballs. The simulation uses the marching cubes algorithm for metaball rendering, heightmaps for terrain construction, and a simple particle system to simulate the physics of the water. The simulation technique is found to have drawbacks with visual plausibility, frame rate in real time performance and scaling. The advantage of the technique is its dynamic and interactive nature.

1 Introduction

The simulation of liquids is a problem in physics and computer graphics with many solutions that seek to accomplish different things. Moving the simulation of liquids into real-time computing and real-time graphics is a challenge of its own. Achieving a physically accurate simulation that runs in real time appears to be impossible at the moment, but there exist contexts where physical accuracy is not the main priority. One such context is that of video games, where plausibility is sought-after, but the main focus is performance.

Devising a technique for simulating water in particular in real time advances the field of video game development, enabling video game developers to realize new game ideas. Water flowing in rivers in real time could be an interesting area for game development, and creating a technique for that type of simulation is therefore valuable.

One approach to simulating water is to use the concept of metaballs. A system of metaballs is used to calculate values in a scalar field, that can then define an isosurface which is rendered as a triangle mesh. The surface that metaballs define is often described as organic-looking. In the present paper, the idea is to let the surface represent water and make the metaballs be affected by forces as if they were water droplets. The aim is to explore how metaballs can be used to simulate water flowing in a river, and to create such a simulation.

1.1 Related work

Previous scientific work have had similar aims as the present paper in some ways, the most relevant of which are summarized in this section.

Müllenmeister in [5] aims to use metaballs to simulate water droplets in real time on low-resource devices like mobile phones. Two types of water droplets are defined. One type is static, and able to merge with other water droplets, with the droplets affecting each other using spring forces. The second type is a water droplet that moves down a surface according to gravity, with its shape changing as a real world water droplet would. To render the metaballs, Müllenmeister uses both an octree technique which is slower, and the marching cubes algorithm which uses more memory. Müllenmeister concludes that simulating water droplets using metaballs is possible in real time on mobile phones.

Kommareddy et al in [4] also use the marching cubes algorithm to render metaballs. The paper

defines useful properties of metaballs and how the marching cubes algorithm uses them. Bourke in [1] gives the original formulation of the marching cubes algorithm.

Chang et al in [2] aim to simulate a river in real time. A heightmap texture is used to define the terrain of the river, and additional textures to define e.g. the flow direction of the river. The simulation uses a GPU-based particle system with two-dimensional metaballs rendered using a billboard technique. Chang et al focus on the splashing behavior of the river water and implement a state machine to handle the splashing of the water.

1.2 Research question

With the aim in mind, and with the previous related work as background, the research question for the present paper is defined:

> *What are the advantages and drawbacks of simulating rivers in real time using 3D metaballs rendered using a triangle mesh?*

Criteria that is taken into account is the visual plausibility (likeness to a real-world river), quality of real-time performance and scalability of the simulation.

2 Implementation

In this section, the implementation of the multiple distinct parts of the simulation are explained. The implementation is done using Unity, making use of the Unity API in C#.

2.1 Metaball rendering

Metaball rendering is done using the marching cubes algorithm. The algorithm polygonizes a scalar field, as described by Bourke in [1], so the first step is to create a scalar field. First, a uniform 3D cube grid is defined by using a Unity axis-aligned bounding box and a resolution. Grid cells are defined as cubes inside the bounding box with a size according to the resolution. Every frame, a value is calculated in each grid point according to the position of the metaballs as in equation 1 adapted from [4]:

$$\text{grid point value} = \frac{1}{n}\sum_{i=0}^{n} f_i(x, y, z) \qquad (1)$$

where n is the number of metaballs, and f_i is the falloff function for metaball i as defined in equation 2, also adapted from [4]:

$$f_i(x, y, z) = 1/((x - x_0^i)^2 + (y - y_0^i)^2 + (z - z_0^i)^2) \quad (2)$$

where (x_0^i, y_0^i, z_0^i) is the 3D position of metaball i.

When each grid point has a value associated with it, the scalar field is complete, and can be used by the marching cubes algorithm. As explained by Bourke in [1], the algorithm steps through each grid cell made up of 8 grid points and compares each grid point's value to a threshold value. Points with values below the threshold are considered to lie within the surface, and all other points are considered to lie outside the surface. Depending on which points of a grid cell lie within the surface, different combinations of triangles are added to the surface's triangle mesh. Bourke presents lookup tables in [1] that define how the triangles should be oriented within the grid cell. These tables, as well as Bourke's description of the algorithm is used to create the procedure that constructs the triangle mesh for the metaball surface in the present paper.

The marching cubes algorithm does not handle triangles sharing vertices, which is a requirement for Unity to be able to recognize how to apply Gouraud shading as in [3] when shading the mesh. Therefore, a hash table (Dictionary in C#) is used to enforce uniqueness of vertices, meaning only one vertex is allowed per unique position in 3D space. Theoretically, the uniqueness is not guaranteed, but it works sufficiently well for the number of vertices considered in this case.

Figure 1 shows the results of the metaball rendering at an early stage in the project. Notice that Gouraud shading is applied, handled automatically by Unity thanks to the triangles in the mesh sharing vertices.

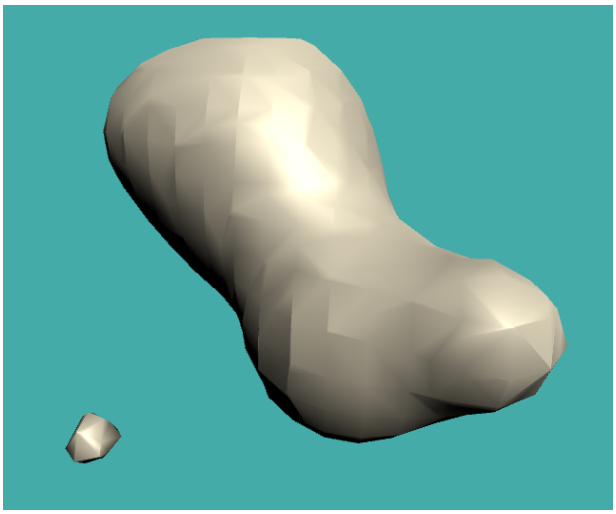

Figure 1: The results of the metaball rendering at an early stage in the project.

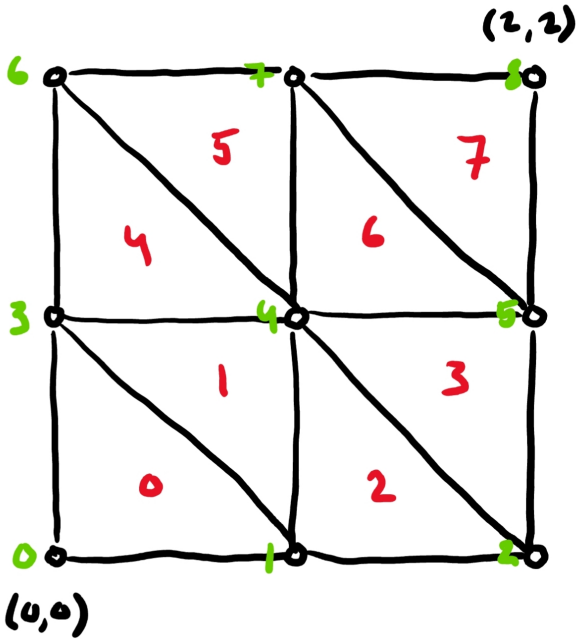

Figure 2: A four-cell example of how the terrain grid defines the terrain triangle mesh. The green numbers are vertex indices, the red numbers are triangle indices.

2.2 Terrain construction

Terrain construction is in this project handled through conversion of a heightmap texture into a triangle mesh, similar to Chang et al in [2]. The heightmap texture is a grayscale image, where all pixel values effectively range from 0 to 1, with 0 as black and 1 as white. The terrain is considered a uniform square grid, where each pixel in the heightmap texture is a grid point. The image position of the pixel is translated into the x and z coordinates of the grid point, and the pixel's value was translated into the y coordinate of the grid point. Both the height scale and the area scale are made into parameters for the terrain, making it possible to adjust the scale.

With the terrain grid defined, a mesh can be constructed from it. Each grid point is made into a vertex, and each grid cell is made into two triangles. Figure 2 shows how each vertex and triangle relates to each other, and figure 3 shows the results of the terrain construction at an early stage in the project.

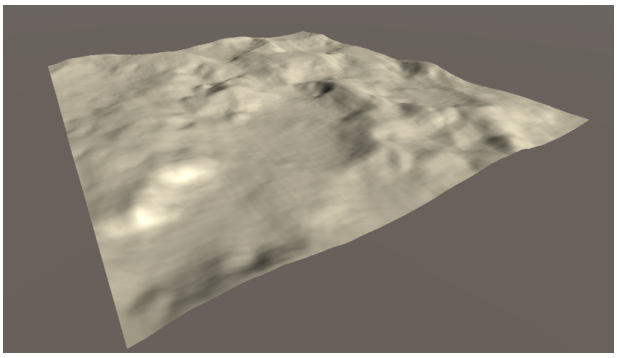

Figure 3: The results of the terrain construction at an early stage in the project.

2.3 Physics and particle system

In order to control the movement of the metaballs, a particle system is implemented. The particles

are metaballs, and the particle system initializes the metaballs in a spawn location and applies forces to them to make them move during the simulation. Three meaningful forces are simulated: gravity, inter-particle spring forces and collision reaction forces. In order to simplify the implementation of the physics system, no actual forces are modeled. Instead, each particle has a velocity associated with it, and the cumulative acceleration caused by all the forces is applied to the velocity each frame. This is made possible by Newton's force equation $\mathbf{F} = m\mathbf{a}$ where $\mathbf{F}$ is the force, m is the mass and $\mathbf{a}$ is the acceleration. By assuming that the mass of each particle is 1 kg, we have $\mathbf{F} = \mathbf{a}$, which means that a force can be calculated, but then applied as an acceleration. Assuming a mass of 1 kg for every particle is arbitrary but reasonable, since mass does not have any purpose in the simulation.

2.3.1 Gravity

Gravitational force is the simplest force in the simulation. In order for the simulation to closely approximate the gravitational force on the surface of Earth, the gravitational acceleration is chosen as $g = 9.82$ $\mathrm{m/s^2}$, applied in the global downward direction. Each frame, the gravity acceleration is added to the cumulative acceleration of each particle for that frame.

2.3.2 Inter-particle forces

Implementing forces between the particles is more complex. The idea is to model a force that ensures that particles stay a certain distance away from each other, while also attracting each other. This is easily done using spring forces, meaning Hooke's law $\mathbf{F}_s = k\mathbf{x}$, where $\mathbf{F}_s$ is the spring force between particles, k is Hooke's constant, or the stiffness of the modeled spring and $\mathbf{x}$ is the vector from the particle to the equilibrium position. In the simulation, the minimum distance and spring stiffness are variables, making the spring force equation take the form of equation 3:

$$\mathbf{F}_s = k(\mathrm{r}\hat{\mathbf{v}} - \mathbf{v}) \tag{3}$$

where r is the minimum distance, $\mathbf{v}$ is the vector from the particle the force is applied to to the other particle contributing to the spring force and $\hat{\mathbf{v}}$ is normalized $\mathbf{v}$. A maximum distance is also defined, beyond which no spring force is applied between particles. Each frame, the spring force is calculated between each particle and each other particle and half the force is applied to each contributing particle in a pair of particles.

2.3.3 Collision detection and reaction

The most advanced force to calculate is the collision reaction force. For this, first a collision must be detected and then the force applied. The collision detection is based on ray-triangle intersection. Each particle holds both a position vector and a last position vector. The difference between the vectors defines a direction and the last position defines an origin. The direction and origin form a ray, with a maximum distance defined by the position vector's distance from the last position. The ray is then checked for collision against each triangle in the terrain's triangle mesh using the typical ray-triangle intersection test in equation 4:

$$\lambda = \frac{(\mathbf{p}_0 - \mathbf{p}_r) \cdot \mathrm{n}}{\mathbf{t}_r \cdot \mathbf{n}} \tag{4}$$

where λ is the distance along the ray to the intersection point, $\mathbf{p}_0$ is some point on the triangle, $\mathbf{p}_r$ is the ray's origin, $\mathbf{t}_r$ is the ray's direction and $\mathbf{n}$ is the triangle's normal. If λ lies between 0 and the maximum distance, and the intersection point $p = \mathbf{p}_r + \lambda\mathbf{t}_r$ lies within the triangle, there was an intersection between the particle and the triangle between the last frame and this frame.

In the case of an intersection, the velocity is dampened by some factor, and reflected along its perfect reflection direction with the triangle's normal. The new position of the particle is the distance from the intersection point the particle should have traveled with its new velocity after colliding with the triangle. This manipulation of velocity and position is considered the collision reaction force.

Testing for collision between every particle and every triangle every frame is computationally expen-

sive. A good approach for significantly lowering the cost of the intersection tests is to implement a tree of axis-aligned bounding boxes (AABB). First, a single AABB encloses the entire terrain mesh. Then, new AABBs are formed as the two halves of the previous AABB recursively, stopping when an AABB contains a small number of triangles, e.g. eight triangles. For this, the heightmap for the terrain is required to have a width and height equal to a power of 2 plus 1, so $2^k + 1$ for some positive integer k, in order to guarantee that there is always an even number of triangles in each AABB so that it can be halved. The AABB tree implementation reduces the number of intersection tests for each particle each frame from at most 65,536 to at most 21 assuming a heightmap of size 256×256 pixels, and eight triangles per leaf node in the AABB tree.

Testing the collision detection reveals a problem. Because of the imprecision of floating point numbers, there are seams between triangles in the terrain mesh that are not visible to humans, but that occasionally let a particle fall through the mesh. This happens most frequently when a particle is moving slowly and encounters a seam. To help solve this issue, particles are offset upwards a small distance on each collision, so that they never move slowly. Despite minor tricks like the constant offset, the particles fall through. After looking into the issue, it seems that there is no simple solution to the problem, and since collision detection is not the main focus of the project, it was decided that the problem would be allowed to persist. Particles that fall through are simply returned to the spawn location after falling out of bounds of the terrain's AABB.

2.4 Unity physics

The problem mentioned at the end of section 2.3.3 was solved with a different approach. The approach in section 2.3.3 should be considered the main approach in the project, and this different approach is simply to achieve more visually satisfying results. This approach does not replace the project's main approach, it supplements it in order to more thoroughly explore the possibility of using metaballs for river simulation.

The Unity engine has a physics engine built into it, which was utilized in this different approach. Each particle is given a sphere collider and a rigidbody component. The terrain mesh is given a mesh collider. The physics engine is able to calculate gravity and collisions, and for simplicity the spring forces are omitted from this approach.

3 Results

In this section, images of the final results of the simulation are displayed. Note that figures 1 and 3 of section 2 are only early, intermediary results. For more comprehensive, animated and interactive results, see section 6.2.

3.1 Project physics

This subsection presents the results of the implementation described in section 2.3.3, that does not use the physics native to the Unity engine.

The final simulation can be described as follows:

1. The terrain mesh and AABB tree for collision detection are constructed.
2. All metaballs are initialized with a random position within a spawning volume.
3. With a given frame interval, metaballs are activated, making them contribute to the metaball rendering and making them part of the system of forces.
4. Forces are applied to the metaballs, making them collide with the terrain and slowly move along the slope of the terrain.
5. Metaballs falling through the terrain or reaching the bottom are reinitialized with a new random position in the spawning volume. Repeat step 4 and 5 ad infinitum.

Figure 4 shows an optimal instant of the simulation, where the river looks plausible. However, figure 5 is more representative of the average instant of the simulation. The behavior shown in figure 5 arises

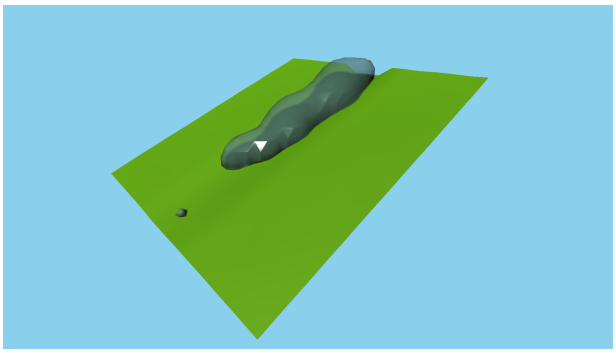

Figure 4: Image from the final simulation showing an optimal instant in regards to river-like visuals.

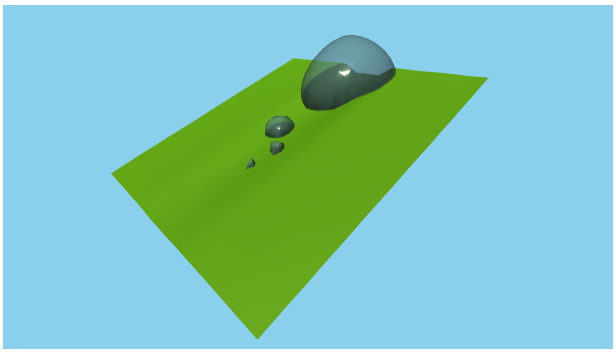

Figure 5: Image from the final simulation showing an instant that is representative of the average instant.

as a consequence of the collision detection issues explained in section 2.3.3, since particles frequently fall through the mesh and respawn at the top of the terrain. The heightmap used for the results in figures 4 and 5 is a very simple heightmap made up of a smooth gradient from black to white with a darker strip in the center of the texture. It is meant to give the forces and metaballs focus.

Figure 6 shows how the spring forces work with the gravity in order to keep a minimum distance between particles. With no spring forces, the particles would all gather in the same position at the bottom of the bowl.

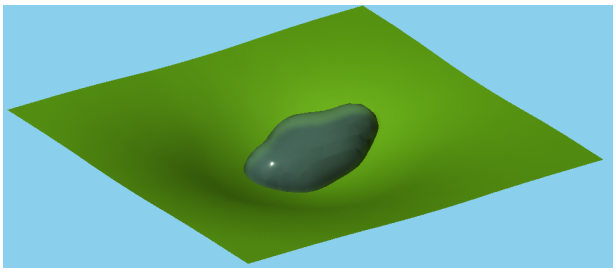

Figure 6: Image from the final simulation showing how the spring forces keep the particles a minimum distance apart.

3.2 Native Unity physics

This subsection shows the results of the implementation that uses the physics native to the Unity engine, as described in section 2.4.

The step-by-step description of the simulation of this version with Unity physics is identical to the one in section 3.1, aside from the fact that the Unity physics engine handles all forces and collisions.

Figure 7 shows the river using Unity physics, which gives a more visually plausible result, even using the same heightmap as figures 4 and 5. Figure 8 uses a different heightmap, one that more closely resembles a real river, and also gives more visually plausible results.

4 Discussion

This section contains a discussion about the results in relation to the research question and criteria.

Starting with the first criterion mentioned in section 1.2, the plausibility of the simulation is discussed. By observation of the figures in section 3, the metaballs are able to simulate a rough appearance of water. However, the simulated river does not resemble a real-world river. The water moves as if it was a small amount of water, e.g. a collection of water droplets. In this way, the metaballs are able to approximate the appearance and behavior of water, but not in the context of rivers. It is possible that increasing the amount of metaballs could improve the visual plausibility, at the cost of performance.

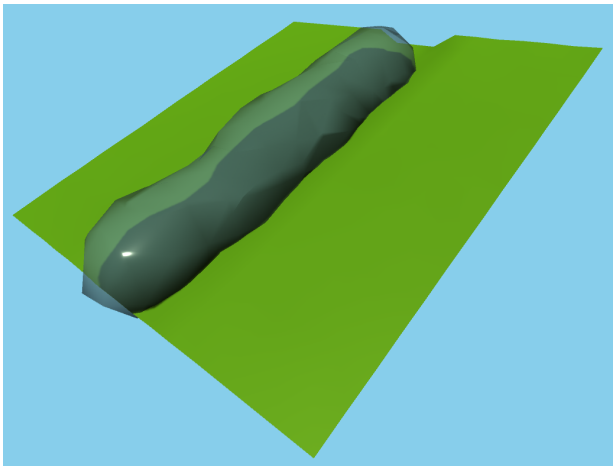

Figure 7: Image from the Unity physics addition using the same heightmap as in figures 4 and 5.

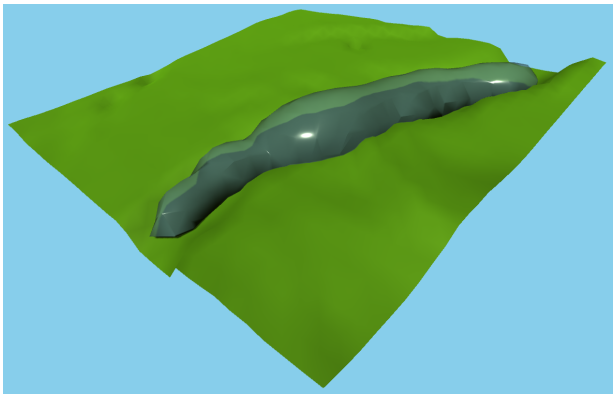

Figure 8: Image from the Unity physics addition using a heightmap from a real river.

A different problem of plausibility is the amount of triangles in the triangle mesh constructed using the marching cubes algorithm. A higher number of triangles (higher number of grid points, higher resolution) creates a smoother surface that more closely resembles real water, and so increases plausibility. Unfortunately, increasing the number of grid points scales badly with the amount of computations required. The marching cubes algorithm asymptotic time complexity is $\mathcal{O}(n_x n_y n_z)$ where n_x, n_y, n_z are the number of grid points in the respective dimension of the cube grid.

The simulations using Unity physics achieves a more plausible behavior, but the results still resemble those of [5] where droplets are simulated. The sought-after behavior is that of the results of [2], which none of the present paper's results resemble.

The low-performance laptop computer that the simulation was developed on is able to run the simulation in figure 6 in real time with frame rates between 30 and 60 fps. All other simulations are reduced to frame rates between 10 and 30 fps, even on a high-performance desktop computer once all metaballs have been spawned.

All simulations presented in section 3 are carried out on a small scale. The number of particles never exceeds 40, the terrain mesh contains a relatively small number of triangles and the resolution of the water mesh is relatively low. As mentioned above, both the number of particles and the resolution of the water mesh need to be scaled up in order to achieve more visually plausible results. However, as was also mentioned above, scaling up the resolution is highly detrimental to real-time performance.

Increasing the number of particles in the simulation requires more forces to be computed and more collisions to be detected. Through informal testing, the number of particles was found to affect performance greatly. As was mentioned above, not even a high-performance desktop home computer is able to run a simulation with 40 particles at high frame rates such as 30 or 60, so the performance scales badly with the amount of particles.

One way of improving the performance and scalability of the simulation is to parallelize the marching cubes algorithm. The algorithm is highly suitable

for parallelization because is runs linearly through a cube grid, and each computation is independent from the others. This parallelization is non-trivial to implement, especially in Unity where so called compute shaders that utilize the GPU are used, so it was not included in the scope of this project.

The terrain construction is handled before the simulation starts, so the performance of the terrain construction does not matter to the overall real-time performance, but the number of triangles does. The number of triangles is $2(n-1)^2$ where n is the width or height of the square heightmap. Increasing the number of triangles slows down the collision detection somewhat. Usage of the AABB-tree lessens the negative effect, but it would likely be noticeable for large heightmaps.

According to all the discussion above, using metaballs for river simulation has a lot of drawbacks. The advantage of metaballs however, is their interactivity and adaptability. Using different heightmaps for the terrain is never a problem, since the metaballs are animated according to physics. If some degree of interactivity was implemented in the simulation, e.g. users having the ability to move the particles, no changes would have to be made to the metaball rendering, and the organic-looking behavior would remain the same.

4.1 Perceptual study

One aim of the simulation is that it should run in real time on lower-end machines. The reason for this is that the simulation method should be possible to use in interactive environments that need to run in real time to be interactive. One such environment is a video game environment. According to Vines et al in [7], visual plausibility is a priority in fluid simulation in computer animation. For this reason, the visual plausibility of the simulation implemented in a video game environment should be evaluated using a perceptual study with human users.

The perceptual study should essentially evaluate whether users accept the water simulation as real water inside the context of the video game environment. This kind of acceptance is described as suspension of disbelief by Piil in [6]. The study would evaluate whether a user, when presented with a video game environment containing the water simulation, would describe the simulated water as water without hesitation.

The proposal for how to conduct this perceptual study is to construct a simple video game environment that users are then allowed to play through. During play, users would be asked to continually describe what they see in a think aloud-style evaluation. The user should not be aware of what is being evaluated. The video game would be designed e.g. as a simple 3D platformer game where the player jumps and runs around the environment to collect points and reach a goal. Something similar to the game Super Mario 64 but with more modern graphics would suffice. Somewhere is the environment, there would exist a river where the water is simulated using the proposed metaball river simulation. The river should be clearly visible, but not critical to the completion of the video game's objectives.

The optimal reaction to the water simulation would be that users describe the river as a river, with just as much importance placed on the description of the river as on any other part of the environment. This would indicate the the user has suspended disbelief, and that the river simulation fits well enough into the environment as to not cause disbelief. The conclusion to be drawn would be that the river simulation has sufficiently high visual plausibility to be used in certain video games.

Users responding the opposite way, dwelling on the description of the river, not being certain what they are looking at, would indicate that there is no suspension of disbelief. The conclusion would in this case be that the visual plausibility is not good enough, either because of the rendering or the physical behavior.

The perceptual study is based on an assumption that it is possible to achieve the results presented in 3 in real time with frame rates of 60 or higher. Another critical assumption is that the graphical style of the video game does not approach photo-realism, and rather limits itself to "cartoonish" exaggeration of real-world phenomenon. Without that assumption, it is unlikely that the the proposed river simulation would fit into the game in any way.

5 Conclusion

The discussion of section 4 practically answers the research question, and the final summarized conclusion is presented in this section. The advantages and drawbacks are summarized in the next paragraph.

The developed technique for simulating rivers in real time using metaballs does not achieve very river-like visuals or behavior. It does however achieve a visually plausible simulation of smaller amounts of water, i.e. water droplets. Because of the marching cubes algorithm as well as the collision detection and spring force computations, the simulation is only able to perform in real time when the particle count is very small. With particle counts over 40, frame rates over 30 frames per second are not achievable, even on medium-performance home computers. The technique does not appear to scale well with the size of the terrain mesh and the particle count.

In summary, the developed technique has drawbacks with visual plausibility, performance and scaling. The only clear advantages are that the simulation is dynamic and interactive, and that the simulated water resembles some types of water, just not rivers. Because of the drawbacks in the current state of the simulation technique, it is not very applicable for use in real-time computer animation of rivers.

5.1 Future work

There are multiple ways of improving the simulation. Parallelizing the marching cubes algorithm, and moving it to be computed on the GPU rather than the CPU, would likely greatly improve performance. Using Unity physics in conjunction with the spring forces would likely also be a good approach to achieving good scaling and performance. The visual plausibility is harder to improve, and some new technique for rendering foam and other river phenomena would have to be devised.

6 References

6.1 Bibliography

[1] Paul Bourke. *Polygonising a scalar field*. 1994.

[2] J.-W. Chang et al. "Real-time rendering of splashing stream water". In: *Proceedings - 3rd International Conference on Intelligent Information Hiding and Multimedia Signal Processing, IIHMSP 2007*. Vol. 1. 2007, pp. 337–340. ISBN: 0769529941.

[3] Henri Gouraud. *Computer display of curved surfaces*. Tech. rep. UTAH UNIV SALT LAKE CITY COMPUTER SCIENCE DIV, 1971.

[4] Sindhu Kommareddy, Jed Siripun, and Jenny Sum. *3D Object Morphing with Metaballs*.

[5] CC Müllenmeister. "Simulation of Water Droplets using Metaballs on mobile phones". MA thesis. 2012.

[6] Markus Piil. *The Suspension of Disbelief Required by Video Gaming*. URL: `https://www.scholarlygamers.com/feature/2017/05/11/suspension-disbelief-required-video-gaming/`. (accessed: 2020-05-04).

[7] Mauricio Vines, Won-Sook Lee, and Catherine Mavriplis. "Computer animation challenges for computational fluid dynamics". In: *International Journal of Computational Fluid Dynamics* 26 (July 2012), pp. 407–434. DOI: `10.1080/10618562.2012.721541`.

6.2 Additional resources

- A blog was updated in parallel with the development of this project. It contains animated demonstrations of different stages of the project, and can be found at the following ULR: `https://metawater.blogspot.com/`.
- The project's source files are available on Github: `https://github.com/zteen/metawater`.