GPGPU: Particle Simulation

Abstract

In this assignment we will learn how to implement two simple particle systems on the GPU. In the first simulation the particles do not interact with each other. The second assignment will then introduce local constraints that will affect the movement of particles to simulate materials such as cloth.

1 Scientific Simulation

Particle systems are widely used to simulate natural phenomena like smoke, fire, or water. In numerical simulation, we distinguish two approaches that conceptually differ in how they handle the simulation domain.

离散化

Lagrangian Approach The Lagrangian approach is based on the discretization of the domain into a set of finite mass elements. These particles are then allowed to move freely, or with respect to defined rules, through the environment. Each particle is usually described by its position and velocity. According to our needs, it can also have any other quantity such as mass, temperature, or radius. Methods based on tracking particles over time and altering their quantities according to the prescribed laws are called particle Systems. Since they directly track chunks of matter through the space, particle systems can easily guarantee the conservation of mass. On the other hand, handling of incompressibility (e.g., in the simulation of fluids) might be more difficult to achieve. 不可压缩性

立体像素

Eulerian Approach Instead of simulating the phenomena using chunks of matter, we can partition the domain into a set of small symmetric cells: voxels. This approach is called Eulerian discretization and the main idea resides in allowing the matter to freely move through the fixed grid, while tracing the simulated quantities. In other words, we do not track the matter itself, but we simulate the quantities at fixed positions in the domain. The discretization is in the simplest case a regular Cartesian grid, however, more advanced techniques often employ hierarchical or multi-resolution structures to devote more computation and resolution to areas, where the quantities change with higher frequency.

交叉点

In this assignment we will use the Lagrangian approach simulating a number of particles that do not interact with each other (first task), and a system where the particles represent a cloth and interact with their neighbors under some constraints (e.g., springs).

1.1 Verlet Integration

Having a system with the matter represented as particles, we want to find out the position of each particle at an arbitrary time. As long as the system is just moderately complicated, we cannot use the closed form solutions and have to solve the equations numerically. There are various approaches to numerical integration that trade speed for quality. We will use a basic integration scheme called Verlet integration.

The *Verlet velocity integration* defines the new position $\mathbf{x}(t+\Delta t)$ and velocity $\mathbf{v}(t+\Delta t)$ of a particle as:

$$\mathbf{x}(t+\Delta t) = \mathbf{x}(t) + \mathbf{v}(t)\Delta t + \frac{1}{2}\mathbf{a}(t)\Delta t^2$$

$$\mathbf{v}(t+\Delta t) = \mathbf{v}(t) + \frac{1}{2}(\mathbf{a}(t) + \mathbf{a}(t+\Delta t))\Delta t$$

The Δt refers to a discrete time step that we take between two (current and the new) simulation steps. $\mathbf{a}(t)$ refers to the acceleration (i.e. first derivation of the velocity, second derivation of the position) of the particle. As a thorough explanation of the underlying motivation for using this integration scheme is beyond the scope of this text, we refer you to other resources for more detailed descriptions of the velocity Verlet integration.

1.2 Collision Detection

Since the particles can interact with the environment, e.g., collide with a wall, we need to simulate these interactions. Consider for instance a particle that is on one side of a wall at time t and on the other side of the same wall at $t+\Delta t$. Obviously, the velocity and acceleration of the particle have moved it through the wall. We should try to detect such interactions and correct the position of the particle accordingly. The most robust approach is to advance only by Δt that is collision free: there is no collision within the whole system until Δt. This is called continuous collision detection, which requires computing the time to the next collision for each particle, and setting Δt to the minimum of these values. In systems with frequent collisions, this can make the time step very small considerably slowing down the overall simulation.

We will take a simplified approach: we always advance by a constant time step and correct for all previously occurring collisions in a post process. Furthermore, we will only account for one collision per particle during a single time step. Therefore, given the old and new particle position, $\mathbf{x}(t)$ and $\mathbf{x}(t+\Delta t)$, we will search for an intersection of the line between these two points and all the objects in the scene. If there is such an intersection, we should correct the position of the particle using one of the approaches shown in Figure 1.

http://www.twinklingstar.cn/2016/2948/4-3-linear-triangle-intersect/

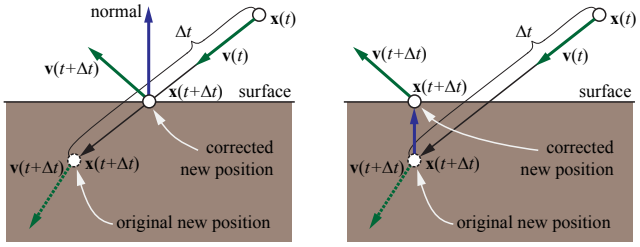

Figure 1: ***Left:*** *We correct the position of the particle by pulling it back where the collision occurred.* ***Right:*** *We take the new position of the particle and project it onto the surface using the surface normal. The corrected velocity is in both cases computed by reflecting the original velocity about the normal of the surface.*

配合，互操作

2 OpenGL Interoperation

During the simulation the particles will move along complex paths in 3-dimensional space and collide with solid surfaces. To evaluate the results, we need to provide real-time visual feedback about the status of the simulation. OpenCL is suitable for general-purpose programming, but we should employ one of the standard rendering APIs to efficiently display 3D content on the screen.

This assignment demonstrates the basics of interoperability. We will use OpenCL to update the state of the simulated world (particle positions, velocities, etc.) and OpenGL to display them. In this case, interoperability means that the same resources will be used in multiple contexts. For example, triplets of floats in an OpenCL buffer memory object can be reinterpreted as vertex positions by OpenGL and displayed as a set of points in space.

OpenGL 2.0 defines buffer objects to hold rendering data. Based on their usage, we can talk about vertex (VBO), pixel (PBO) and texture (TBO) buffer objects. If we want an OpenCL kernel to be able to modify the rendered geometry, we can create a buffer memory object from a VBO. As OpenCL and OpenGL coexist on the same device in parallel, there can be conflicts when accessing the shared resources. Therefore, before executing OpenCL kernels that use the shared buffers, the OpenCL must place a lock these buffers.

As graphics programming using OpenGL is not covered by our course, the implementation of the rendering part will be already provided. To complete this assignment, you only need a basic understanding about the OpenGL context sharing.

3 Task 1: Simple Particle System

In this task we will implement a simple particle system that is driven by a force field. As there are no interactions or collisions between particles, the algorithm can be trivially parallelized using one thread for each particle. We will use a 3D vector field to define the force field within the simulation domain. This force field is loaded from a file and uploaded to the GPU as a 3D texture. Using a 3D texture instead of a regular linear array has two advantages: first, we can use a 3D vector to conveniently address the 3D texture, second, the hardware can automatically perform a trilinear interpolation of the eight nearest neighbors. In other words, if we address a point within the texture that is not exactly one of the discrete positions at which the texture samples the signal, we would have to load the eight nearest neighbors and perform a trilinear interpolation manually. As this is a frequently used operation in rendering, GPUs have a hardware implementation that significantly speeds up the trilinear filtering.

3.1 Integration and Collision Detection

Each particle in our system is defined by a position, velocity, mass, and remaining life. The first two characteristics are 3-component vectors, whereas the mass and life are scalars, therefore, we can pack all the particle data into two `float4` arrays. This will enable coalesced accesses and minimize the fillrate of the application.

In order to implement the Verlet integration and collision detection, add your implementation in the `Integrate` kernel in `ParticleSystem.cl`. This kernel should perform the following steps:

- Load the particle data
- Perform the Verlet integration
- Check for collisions
- Kill old particles
- Possibly create new particles
- Store new particle data

3.1.1 Integration

In order to compute the acceleration of the particle, use the gravitational acceleration and the acceleration defined by the mass and the force at the current position of the particle. For fetching the force from the 3D texture use the function `read_imagef(gForceField, sampler, lookUp)`, where the `gForceField` and the `sampler` are parameters that the kernel obtains and the `lookUp` is a `float4` with the first `xyz` components specifying the position and the `w` defines the mip level (in our case it should be set to 0). These are already initialized to perform the trilinear interpolation.

To be compatible with OpenCL 1.0 we use `float4` vectors. Later OpenCl specifications also support `float3`, so you can also use them if you wish. Take care when performing arithmetic operations on the `float4` vectors that the last component is correctly set to 0.

3.1.2 Collision Detection

In order to compute the collision of particles with the scene objects, the kernel is also given a pointer to a global array with all triangles in the scene. The triangles are stored as a *triangle soup*: the triangles are stored in a consecutive chunk of memory, each triangle is represented by three vertices, and each vertex is defined as a `float4` variable. Given the old and the new position of the particle, you will construct a ray segment and try to intersect this segment with each of these triangles. The straightforward solution is to iterate over all vertices in the global memory, construct a triangle, and call a function that will determine whether the ray segment intersects the triangle.

Since all threads are iterating over the same values, there is a great chance to use the local memory to cache the triangles. One of the possible approaches is to read one vertex from the triangle soup by each thread and store it in the local memory. Then all threads can iterate over the cached triangles in the local memory and perform the collision test. If the number of triangles is higher than the number of cached triangles, we have to repeat the process of loading and testing the triangles multiple times. Notice, that if the number of threads is not a multiple of 3 (the number of vertices for one triangle), it can happen that we will not read the whole last triangle: the threads might load only one or two vertices of the triangle. Unless we want to take a special care of this case, we have to make sure that the number of threads within a work-group is divisible by 3.

We provide you with a `LineTriangleIntersection` function, which computes an intersection of a line with a triangle. Your task is to call this function for each triangle and find the closest intersection, as the particle should collide with the nearest triangle. Once you have it, you can adjust the new position and velocity of the particle using one of the approaches illustrated in Figure 1.

3.1.3 Removing and Adding Particles

After you account for the collisions, you should decrease the (remaining) life of the particle. If the life is smaller than zero, the particle should be removed and you will create a new particle initialized with some default position, velocity and life. Push the limits of your imagination and create some nice simulation (e.g., a monkey spitting particles).

3.2 Implementation Details

The skeleton of the assignment already performs most of the initialization that you will need. It creates the force 3D texture, initializes

强制的

the particle data with some random positions, mass, and life. It also prepares most of the device arrays that you will need. The provided structure of kernels is not mandatory, so you can adjust it to suit your implementation better. There are two scene files: `CubeJump.obj` and `CubeMonkey.obj` that contain a box with some additional geometry. **You can select the actual scene that is loaded in the constructor of the `CAssignment4` class**.

3.3 Visualization

精灵
四方院子

A particle simulated by the OpenCL kernel is mapped to a single vertex during OpenGL rendering. We render the particle system by generating a *sprite* at the location of each particle. A sprite is a quad, which always faces towards the camera. These sprites are then rendered with additive blending to the screen, giving the impression of small, point light sources (Figure 2).

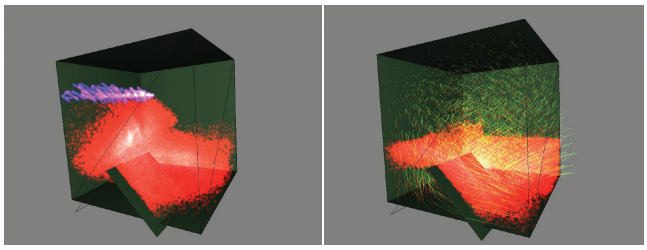

Figure 2: ***Left:*** *192K particles are simulated in a force field, colliding with the surrounding geometry. The color coding shows the dynamically born particles, generated using speed and height criteria, at the top left corner of the image.* ***Right:*** *The force field is visualized using line primitives as indicators.*

To better understand the behavior of the particles, we can modulate the color of the rendered sprites as a function of their mass, life, or speed. This color coding can not only create interesting, colorful visualizations, but can also help for debugging the code. For example, coloring the particles based on their life will clearly show newly generated particles in order to check their behavior. We do not get into the details of visualization, but we explain the color-coding mechanism of the vertex shader, `particles.vert`:

```
uniform samplerBuffer tboSampler;

void main() {

  // get position and life (bound as a vertex buffer)
  vec4 v = vec4(gl_Vertex);

  vec4 position = vec4(v.xyz, 1);
  float life = v.w;

  // get speed and mass (bound as a texture buffer)
  v = texelFetchBuffer(tboSampler, gl_VertexID);

  vec3 speed = v.xyz;
  float mass = v.w;

  // render the particles using their speed
  gl_FrontColor = colorCode(length(speed));

  // [...]

  gl_Position = gl_ModelViewProjectionMatrix * position;
}
```

This GLSL code (the standard shading language of OpenGL) gets executed for each particle vertex. Each vertex will be replaced by a point sprite later on, but the vertex shader will define the position and color of the sprite. The `gPosLife` buffer is mapped as a vertex buffer, so the shader can get the position and life of the particles using the `gl_Vertex` built-in variable. The `gVelMass` buffer is used as a texture buffer (TBO), and fetched using the `texelFetchBuffer()` command. This particular sample then color codes the particle using the length of its speed vector. The last line of the shader is a standard OpenGL transformation from world space (where the particles are simulated) to another coordinate system which OpenGL uses to project the particles to the screen. Feel free to modify this shader to develop different meaningful visualizations of particle behavior.

The application also displays the force field as a set of colored lines. These lines are scattered inside the volume, showing the direction and magnitude of the forces at sample locations. You can toggle the rendering of the force field during simulation by hitting 'f'.

Note that the application loads the collision geometry from a Wavefront OBJ file. That means, you can experiment with different collision objects by replacing the input file to a custom one (see `CAssignment4::CAssignment4()`). As you increase the number of triangles in the collision object, you will notice that the performance of your simulation will drop dramatically. This is because each particle is tested against each triangle in the scene. Using a regular grid to spatially index the triangles based on their position would make the search for collisions a lot more efficient. You can think about such extensions for the free-style assignment.

3.4 Evaluation

The total amount of points reserved for this task is 10:

- Verlet velocity integration (4 points).
- Collision detection and handling of particles with triangles (2 points) cached in the local memory (2 points).
- Aging of particles (1 point).
- Recreation of particles (1 point).

4 Task 2: Cloth Simulation

Real-time cloth simulation is a popular physically motivated simulation even in computer games. In this assignment we will learn how to implement a basic cloth that uses a spring model to maintain the structure of the material and achieve the cloth-like behavior. The springs (or constraints) are used to mimic the thread-like structure of the material. The model is easy to parallelize, however, the resulting cloth behaves somewhat more elastic than most of the materials we know from the real life. Despite this drawback, it gained popularity in game industry, since a basic cloth can be simulated only with a few particles and springs.

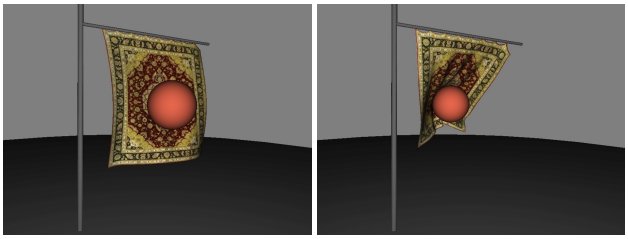

Figure 3: *Cloth simulation based on an explicit model defining the structure via virtual springs between the particles.*

The implementation of the cloth simulation requires the following kernels:

- Integration kernel
- Constraints kernel
- Collision detection kernel
- Kernel for recomputing normals

谨慎地

Each of the kernels is detailed in the following sections. In our implementation, we first execute the integration kernel and then enter a loop that launches the constraints and collision detection kernel several times. In the end we recompute the normals to achieve more correct shading of the cloth. The suggested order and structure of kernels is not mandatory. You can adjust the kernels (for instance by concatenating some of them into a single one), number of iterations, how often you check for collisions, or the data that you exchange in between kernels, if you can justify your decision by meaningful arguments.

4.1 Integration

The first executed kernel is responsible for moving the particles according to all external forces (e.g., gravity or wind). In contrast to the previous task, we suggest using the *Verlet position integration*, which allows handling the correction of the position due to constraints more conveniently. Since we want the cloth to behave like it was attached on top to a bar, we should not trigger the computation for some of the particles in the very first row. This condition is already provided in the code. Notice that the same condition should also appear in the kernel for satisfying the constraints.

4.2 Satisfaction of Constraints

In order to simulate the cloth, we will use a set of constraints that will act as springs between the particles. These springs will try to preserve the initial spacing between particles, pushing them together if they are too far, and pulling them away if they are too close. Figure 4 shows three different types of springs that we will use. The *structural* constraints preserve the original adjacency, *shear* constraints support the grid-like structure, and the *bend* constraints affect the stiffness of the cloth.

弹簧 修剪 弯曲 坚硬

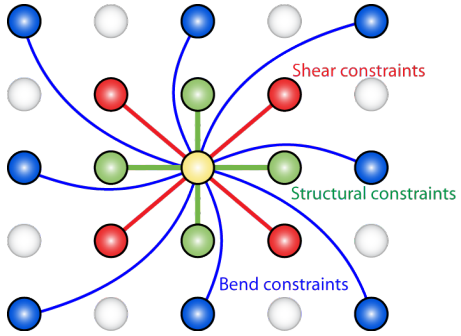

Figure 4: *Different types of constraints (springs) that we use to preserve the structure and cloth-like behavior.*

The kernel for satisfying the constraints should be executed multiple times. During each iteration, we shift the particles a little in order to relax the constraints to get closer to the state with minimum energy. Since we cannot satisfy the constraints in one pass, we have to take an iterative approach and trigger the satisfy constraints kernel. Since all the constraints are evaluated in parallel, we must not move one particle by more than $d/2$ during one iteration, where d is the distance between two particles in the rest state. If we did, two neighboring particles could possibly get attracted or repulsed too much and the computation would "blow up". Therefore, we have to weight the contribution of each appropriately, or clamp the change in position to $d/2$.

拒绝 固定 描述

Figure 5 depicts particles that act on a single particle in a horizontal direction only. Since we do not want to offset the particle by more than $d/2$, we need to conservatively add the forces; otherwise the simulation might blow up. We included some default weights in the code (`WEIGHT_ORTHO`, `WEIGHT_DIAG`, etc.), which you can use to scale the contribution of each spring. Feel free to adjust the values or to use a different approach. You can also adjust the bending and shear constraints to change the stiffness of the cloth.

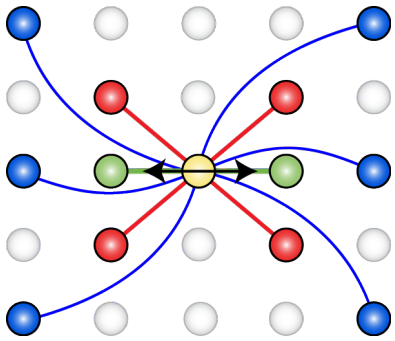

Figure 5: *Neighboring particles acting on the center particle in the horizontal direction only.*

In order to efficiently implement the constraint satisfaction efficiently, you should use the local memory in the same spirit as in the non-separable image convolution assignment. In this case the halo region will correspond to a two particles wide ring around the processed tile.

4.3 Collision Detection

If you run the simulation at this point, the cloth should perform a realistic waving motion according to the applied forces but keeping the distances among particles by satisfying the cloth constraints.

The only step left to finish the simulation part of the task is to make the cloth interact with the solid sphere placed under the bar. To resolve particle collisions with a sphere, you can use the approach illustrated in Figure 1, right depiction.

The collision detection with the sphere is simple: we have to determine if any particle is closer to the sphere center than the sphere radius. In case of a collision, we can push the particle to the surface of the sphere using a vector in the radial direction. Note, that this simple implementation of collision response will always keep the particles on the surface of the sphere. Compared to the collision detection of the first task, this approach is not very robust: it can happen that the sphere travels through the cloth if you push it hard enough. You can improve the detection akin to the previous task, 类似的 but it is not mandatory (to add it should be straightforward).

4.4 Evaluation

The total amount of points reserved for this task is 10:

- Verlet integration accounting for gravity (3 points).
- Satisfaction of constraints (2 points for a solution without using local memory and additional 3 points when using local memory for caching.)
- Collision detection with a sphere (2 points).

There are several options to enhance the simulation. One of them is for instance ripping of the cloth: when the springs between particles stretch too much, they break and are not considered anymore. You can gain up to 2 extra points for implementing this behavior.

Note: In the simulation you can move the sphere by moving the mouse with the third button pressed.