Nvidia FleX

Soft bodies

Graduation Work

Digital Arts & Entertainment

Simon Coenen

Table of Contents

1.0 Introduction

Nvidia FleX is a particle-based simulation library that is based on Position-Based Dynamics and Unified Particle Physics for Real-Time Applications (Miles Macklin). Every object is a system of particles and are all connected by different constraints. Each frame, the solver calculates a position for each particle in the system depending on several interactions and parameters. Using this system, you can achieve various results like cloth, fluids, gases, rigid bodies, soft bodies…

From the manual: FleX is a particle based simulation technique for real-time visual effects. Traditionally, visual effects are made using a combination of elements created using specialized solvers for rigid bodies, fluids, clothing, etc.

The goal of this research is to get to know the Nvidia FleX framework and learn how such a system is built up using only particle physics to eventually implement soft body physics. To start with, I analyzed the demo from Nvidia that is included with the source development kit thoroughly. The application is quite interesting and has several demos to showcase all the Nvidia FleX features in C++ with a very basic OpenGL rendering setup. The demo application and its source code are going to be my main reference for this research.

Like every SDK, Nvidia FleX has documentation that consists out of a short 'Getting Started' manual and a compact code reference document. These documents put you on the right tracks to start out with but unfortunately lack detailed information.

There are integrations of Nvidia FleX available for Unreal Engine and Unity Engine. Nvidia has a custom Unreal Engine branch with many Nvidia GameWorks tools integrated including Nvidia FleX. These integrations are accessible for developers for free on GitHub but the integration is very engine specific, meaning that it is hard to read and specifically adapted to work in Unreal Engine. Unity has an unofficial integration called 'uFlex' and it is available on the Unity Asset store for an acceptable price. Both these integrations are very interesting but I try to stay away from those because it is more interesting to implement it from the bottom up.

2.0 Problems at first sight

The first and most obvious thing to do when starting to research something is search for some possible references and uses. I was surprised to see that there is very little to no information available about Nvidia FleX. The framework is currently being used in only two games: Killing Floor 2 and Fallout 4. Looking at the official Nvidia website, I realized that even the documentation on their website is outdated. The framework is currently in version 1.0 while the documentation is still set on version 0.8. Fortunately, I noticed there is a Nvidia Developer Forum that seemed quite active and tried asking a few questions but after a few weeks, I didn't receive any response.

Luckily the demo included with the SDK is quite extensive and uses almost all the features of the framework except those from version 1.0. This application has very useful demos but it is extremely hard to read since it is specifically made for demo purposes. The rendering API used in the demo is OpenGL while engine I work with is using DirectX so it required me to look in to some OpenGL to understand what was going on in several situations.

3.0 The building blocks of Nvidia FleX

All of the applications of FleX like soft bodies, fluids, cloth, … have the same base principles so it was best to first look at the base building blocks of the framework and then move on the actual creation of soft bodies.

So before digging into the actual implementation of the framework, I carefully analyzed the demo of Nvidia. Since the code is very hard to read, it took some time to figure out how the systems are initialized, the particles are created, updated and then rendered. Below, I will explain the steps to initialize and use the framework.

3.1 Initializing and updating FleX

The FleX system works with different solvers that all act independent from each other. Multiple solvers can be created and updated separately. This makes having multiple effects much easier because it provides more flexibility when setting up the parameters that control the behavior of the particles controlled by a solver.

In the newer version of Nvidia FleX, you can create containers. Containers have basically the same behavior as a solver but manages all the memory allocations their selves. Unfortunately, I found no information whatsoever about this data structure.

Below is a simple overview of initializing FleX with one solver. A more detailed overview of the pipeline can be seen on the image in chapter 3.4.

```
flexInit(FLEX_VERSION);
FlexSolver* pFlexSolver = flexCreateSolver(1000, 0);
while (true)
{
	GameLoop();
	flexUpdateSolver(pFlexSolver, 1.0f / 60.0f, 3, nullptr);
}
flexDestroySolver(pFlexSolver);
flexShutdown();
```

It is interesting that the solver itself does not have any methods but merely acts as a pointer to a set of data that the framework is managing. For every method that alters particles, the solver has to be passed as a parameter. This is the case for many data structures in the framework.

3.2 The particle system

The main idea of FleX is that the base of every object consists out of particles. For each particle, the position, velocity and inverse mass is stored and saved in large arrays managed by the FleX solver.

```
struct Particle
{
	float x, y, z;
	float invWeight;
};
```

The particle structure would look like this but using a type like Vector4/XMFLOAT4 (=16 bytes) would be the same. Like with cloth in Nvidia PhysX, the inverse mass is stored together with the position.

Aside from these properties, each particle has a 'phase'. This phase influences the way particles interact with each other. A phase holds the particle group and the particle flag. The group acts like a collision filter.

Knowing this, you can already set up a simple particle system

```
flexSetParticles(m_pFlexSolver, particles.data(), particles.size(), eFlexMemoryHost);
flexSetVelocities(m_pFlexSolver, velocities.data(), velocities.size(), eFlexMemoryHost);
flexSetActive(m_pFlexSolver, indices.data(), indices.size(), eFlexMemoryHost);
```

As said, when modifying particles, the first parameter is almost always the solver. The last parameter is an enumeration to indicate the memory location of the data. I will elaborate on this later.

3.3 FleX parameters

The behaviors of the particles are set on a per-solver basis. All of these parameters are stored in the FlexParams data structure. The FlexParams can influence for example the gravity of the particles, the friction, the collision distance, ... Important to know is that each solver has its own collision detection pipeline. This means if I needed interaction between systems that need to have different parameters, it would not be possible. Below are the most important parameters.

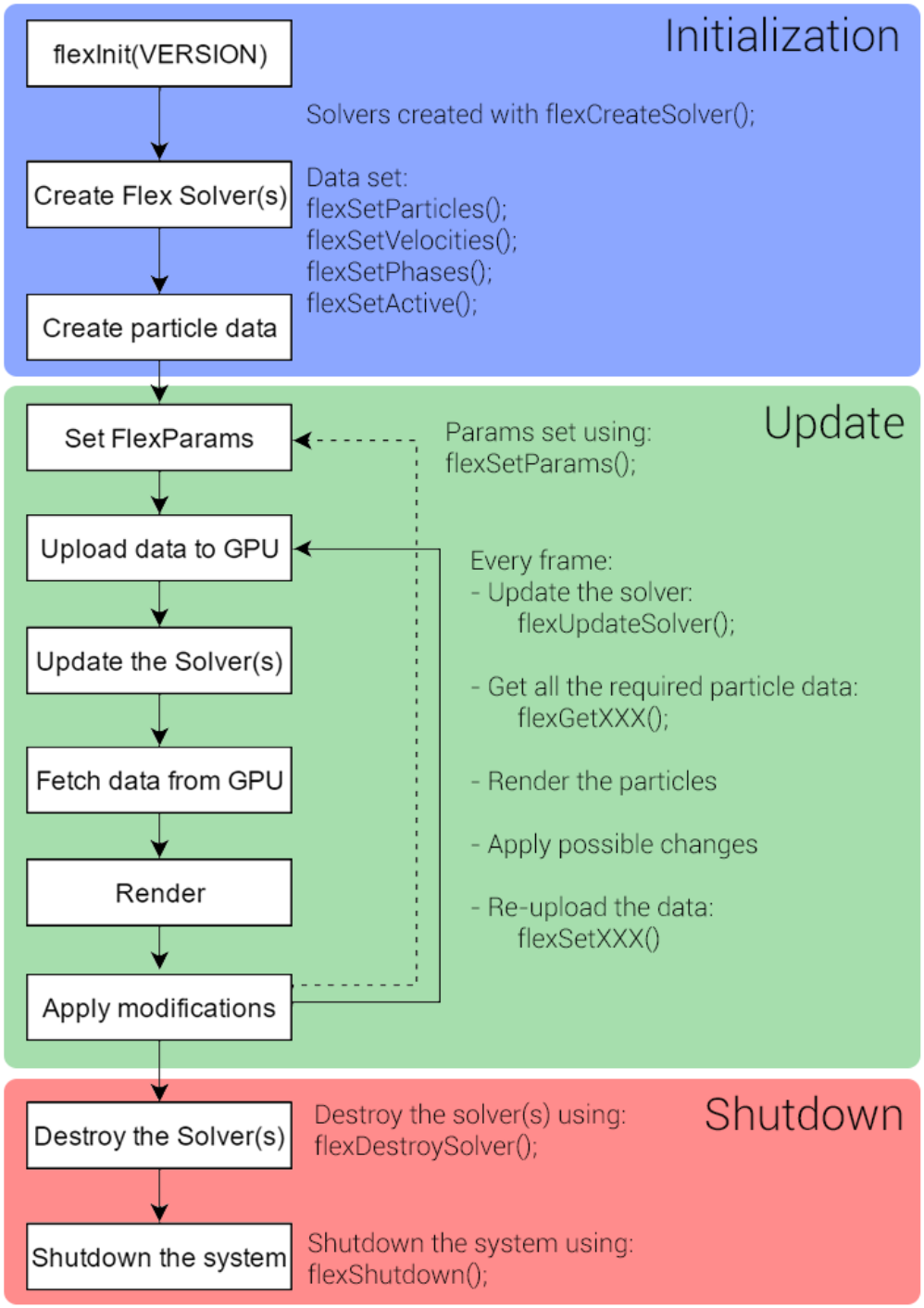
int	mNumIterations	Number of solver iterations to perform per-substep.
float	mGravity[3]	Constant acceleration applied to all particles.
float	mRadius	The maximum interaction radius for particles.
float	mSolidRestDistance	The distance non-fluid particles attempt to maintain from each other, must be in the range]0, radius].
float	mDynamicFriction	Coefficient of friction used when colliding against shapes.
float	mStaticFriction	Coefficient of static friction used when colliding against shapes.
float	mParticleFriction	Coefficient of friction used when colliding particles.
float	mCollisionDistance	Distance particles maintain against shapes, note that for robust collision against triangle meshes this distance should be greater than zero.

The parameters are set simply by using this method:

```
flexSetParams(pFlexSolver, &FlexParams);
```

3.4 The result

By setting up all the things I've said before, I already had a simple application running that had particles behaving like described in the FlexParams. In general, the pipeline can be seen like the image below:



FleX does not take care of any rendering so after creating all the particles, they will update each frame but visually you won't see anything happen. So before continuing studying the FleX framework, I had to make sure I had a way of visually debugging the particles by rendering them as spheres.

4.0 Rendering the particles individually

In the demo of Nvidia, the particles are rendered as spheres, this looked like the best way to visualize them and see how they behave.

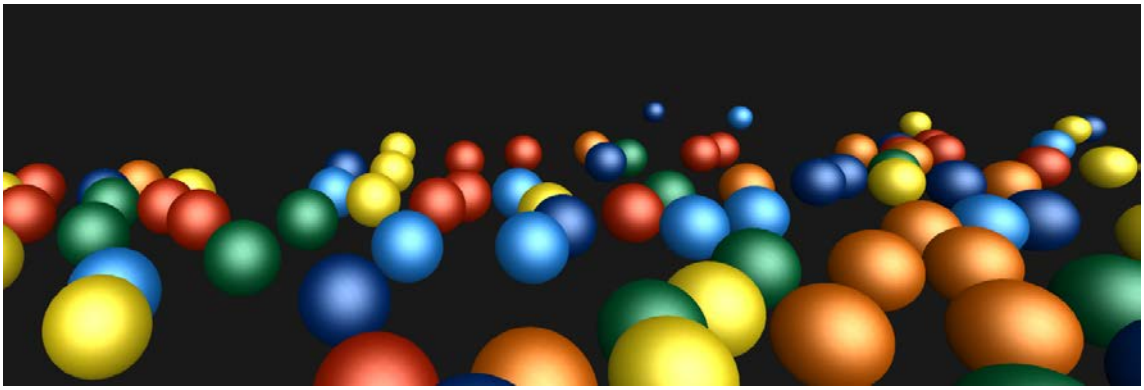
When using the framework, there will be an extreme amount (>10 000) of particles so rendering each particle in a separate draw call would be extremely inefficient. My solution was to make use of the instancing feature of DirectX. I treat every particle as a position so I can upload the position data to the GPU and render them all at once.

The input layout eventually looked like this:

```
//Input Slot 0 (Vertex Data)
{ "POSITION", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 0, D3D11_INPUT_PER_VERTEX_DATA, 0 },
{ "NORMAL", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 12, D3D11_INPUT_PER_VERTEX_DATA, 0 },
//Input Slot 1 (Instance Data)
{ "WORLDPOS", 0, DXGI_FORMAT_R32G32B32_FLOAT, 1, 0, D3D11_INPUT_PER_INSTANCE_DATA, 1 },
{ "PHASE", 0, DXGI_FORMAT_R32_SINT, 1, 12, D3D11_INPUT_PER_INSTANCE_DATA, 1 },
```

The phase of the particle is used to color-code the collision groups of the particles.

To test this, I created some particles with random collision groups (see below). The rendering of the spheres itself is very efficient.



5.0 Data storage

Since the framework does not provide any means to store the data outside the simulation, I needed to created a way to centralize all this data and contain it within one system. Not doing this would make it really difficult to pass around data to different systems and bodies.

I created a FlexSystem class that represented a FlexSolver with all its necessary data like the positions, velocities, collision shapes, rigid data and parameters. This means that instead of creating a FlexSolver for a new system, I create a FlexSystem so it can easily accommodate all possible data required by the system. This turned out to work very well since multiple systems of particles could make use of one single FlexSystem because creating for example a soft body requires you to pass a FlexSystem in the constructor. When I would need a system with different parameters, I simply created a second FlexSystem for that to be updated separately. The FlexSystem updates the solver, fetches its updated data and uploads the modified data at the end of each frame.

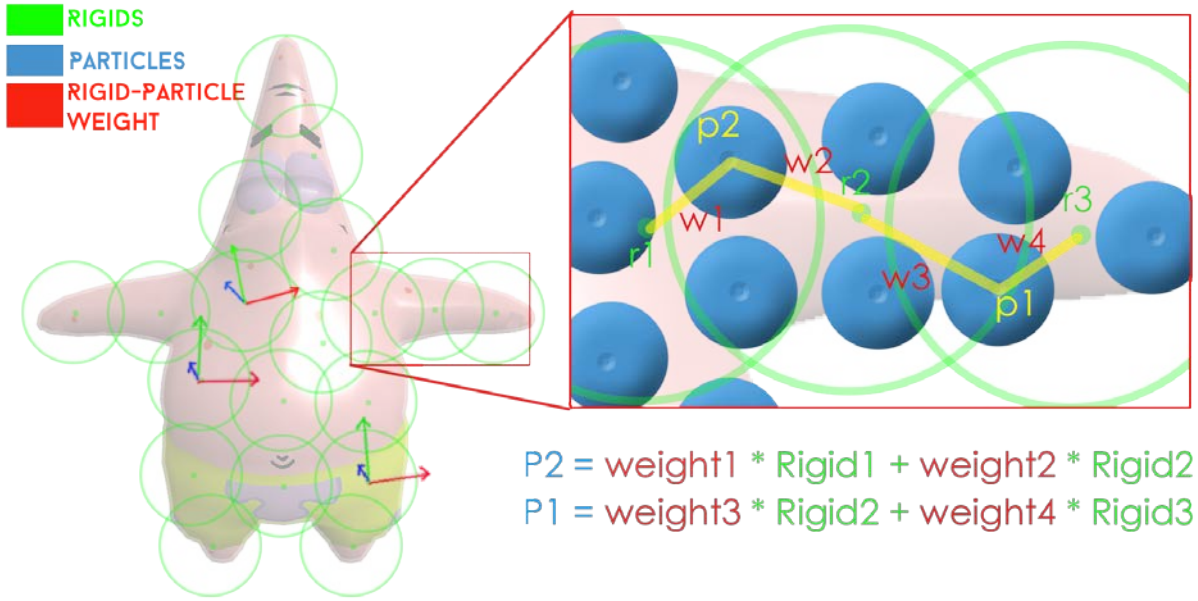
6.0 Starting out with soft bodies

Knowing the basics of FleX, I could move on to looking in to the creation of a soft body. The principle of having position, velocity, phase, … data will stay the same. The way they are obtained is different. On top of that data, a new set of data is introduced: the 'rigids' or 'clusters'.

6.1 The anatomy of a soft body

In FleX, a soft body consists out of several rigids/clusters. Every cluster has their own position and rotation and serve as some kind of 'joint' for the body. Every particle is connected to one or more clusters with a weight between 0.0 and 1.0. Just like skinning for animations, it defines how much a cluster influences the transform of a particle. This means that if you would want a rigid body, there would only be one cluster and all the particles would be connected to this cluster with weight 1.0.

As an example, take Patrick Star on the image below. The green dots represent clusters. Each cluster has a radius (the green circle). If the clusters are overlapping, they are connected by a user-defined stiffness.

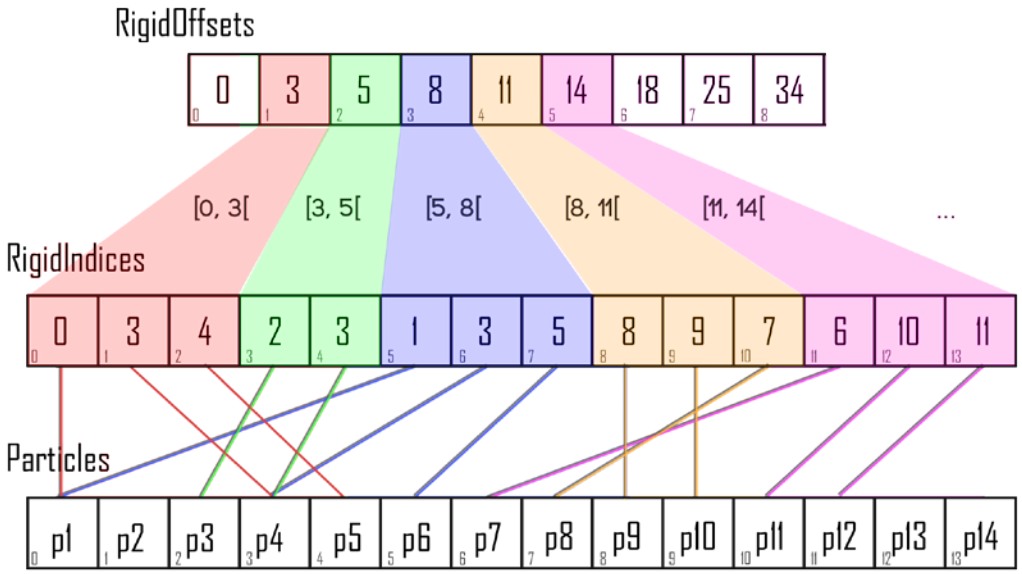


When there is a collision or an external force, the clusters will be influenced by it and because the particles are connected with these clusters, so will they. This is the base principle of rigids/clusters in FleX.

In FleX, this data is stored in three different arrays:

- RigidIndices: Stores the index of the particles that are connected with a certain rigid.
- RigidOffsets. Stores the range of what indices belong to the rigid.
- RigidCoefficients: Stores the weight of each particle to their rigids.

Below an image that shows how these two arrays are related to each other and to the particle data array.



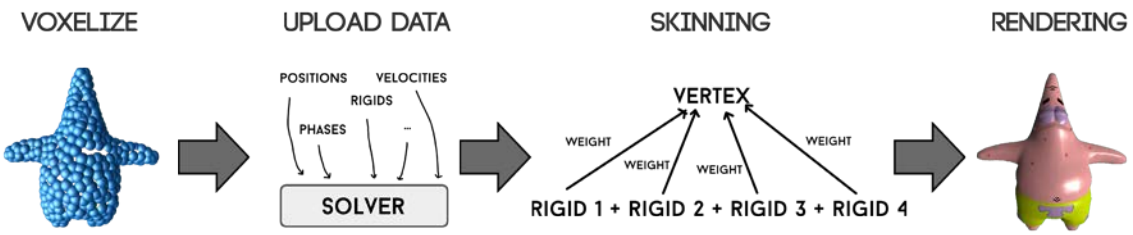
At the start, this seemed very confusing but after creating a diagram like above, everything became clear.

The RigidOffsets always start with '0'. This is something that has to be added explicitly or the program will result in to some hard to find bug. I will explain the system by means of an example using the diagram on the top:

Take rigid with index 4. The rigid offset of that is 8 and the one after is 11. This means that the start offset of the RigidIndices for that rigid is 8 and end at 11 exclusive ([8, 11[ ). Those RigidIndices are: 8, 9 and 7. These indices point at particles p9, p10 and p7. As a result, particles p9, p10 and p7 are connected with the rigid with index 4. The weight of these connections are stored in the Rigid Coefficients array and obtained by the RigidIndices as well.

Knowing this, I could move on to practically creating a soft body. I will split up the process is in to four parts:

1. Voxelizing the mesh
2. Uploading the data to the solver
3. Creating the skinning of the body
4. Rendering the body.



6.2 Creating a soft body

During my research, I started with a basic DirectX 11 framework with a MeshFilter class. This class holds all the vertex information needed to send to the GPU when rendering like the position, normal, color,… The FleX framework has a convenient helper function 'flexExtCreateSoftFromMesh()' that takes care of voxelizing the mesh according the given parameters. The function takes position and index information from a mesh together with a few parameters. These parameters are:

float	particleSpacing	The spacing between the particles
float	volumeSampling	The resolution the mesh is voxelized at to generate interior sampling
float	surfaceSampling	The amount of samples that is taken from the surface to improve the surface detail.
float	clusterSpacing	The spacing for shape-matching clusters] radius, MAX_FLOAT [
float	clusterRadius	The actual size of the clusters. The higher this value, the more the clusters overlap and the more rigid the body will be. If this value is too low, the body will just fall apart
float	clusterStiffness	The stiffness between the connected clusters
float	linkRadius	If particles are below this distance, a distance constraint will be created between those particles with a user-defined stiffness (linkStiffness).
float	linkStiffness	The stiffness of the distance constraints.

The function returns a FlexExtAsset*. This is a structure that contains the particle data that has to be sent to the solver and shape information that is required to set up the skinning for eventually rendering the body. At this point, we already have all the information we need to set up the soft body. It is important that the FlexExtAsset is explicitly destroyed after being used by calling flexExtDestroyAsset() with the pointer to the object as a parameter.

6.3 Uploading the data to the solver

Creating the data for the soft body was quite simple because of the helper function. Getting the data to the solver requires a little bit more work.

As I said, FleX uses clusters/rigids to define the behavior of a soft (or rigid) body. This data is sent to the solver on top of the particle data using flexSetRigids(). This function requires several parameters:

FlexSolver*	s	The spacing between the particles
const int*	offsets	Pointer to an array of start offsets for a rigid in the indices array, should be numRigids+1 in length since the first entry must be 0
const int*	indices	Pointer to an array of indices for the rigid bodies.

const float*	restPositions	Pointer to an array of local space positions relative to the rigid's center of mass (average position)
const float*	restNormals	Pointer to an array of local space normal.
const float*	stiffness	Pointer to an array of rigid stiffness coefficents, [0.0, 1.0]
const float*	rotations	Pointer to an array of quaternions.
const float*	translations	Pointer to an array of translations of the center of mass
int	numRigids	The amount of rigids/clustes
FlexMemory	source	The memory space of the buffer

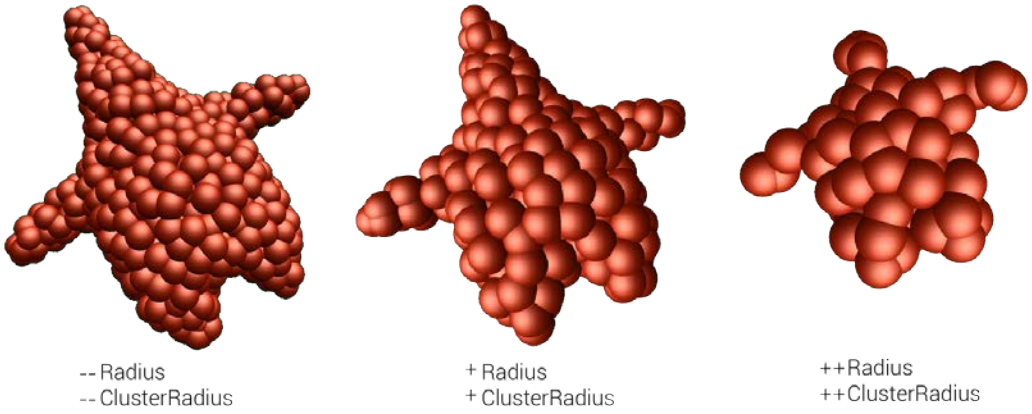
Some of this data we already have, the only thing that still needs to be calculated are the RestPositions. Those have to be in local space relative to the transform of the cluster. The RestNormals are in the case of soft bodies not necessary so this can be set to 'nullptr'.

To get the RestPositions, we have to calculate the position of each particle relative to the average position of the particles that are connected to that cluster, that is the center of mass. Below, the code to acquire this data.

```
int count = 0;
for (int r = 0; r < numRigids; ++r)
{
	const int startIndex = RigidOffsets[r];
	const int endIndex = RigidOffsets[r + 1];
	const int clusterSize = endIndex - startIndex;
	Vector3 average = Vector3();
	for (int i = startIndex; i < endIndex; ++i)
	{
		const int rigidIndex = RigidIndices[i];
		average = average + Vector3((float*)&Positions[rigidIndex]);
	}
	average = average / float(clusterSize);
	for (int i = startIndex; i < endIndex; ++i)
	{
		const int rigidIdx = RigidIndices[i];
		localPositions[count++] = Vector3((float*)&Positions[rigidIdx]) - average;
	}
}
```

For each cluster, I iterate over its indices twice. The first time is to calculate the average position of the particles connected to that cluster. When I have that, I can subtract that average from each particle's positio. This results in the local position of the particle according to the center of mass. Having this data, I can complete the flexSetRigids() method.

At this point, I already had the soft body working and using the particle renderer I wrote for debugging, I could already see a clear result as seen on the image below. You can see how the parameters of the soft body influences the particle density on the image below. Lowering the radius increases the amount of particles and quality of the body but significantly decreases the performance of the simulation. This is of course no final result since now remains the rendering of the actual mesh.



6.4 Creating the skinning

Before I could start rendering the body, I first had to calculate the skinning of the mesh on the particles. This is done by using another convenient method from the FleX extension library: flexExtCreateSoftMeshSkinning(). As the method for creating the soft body itself, it requires the position data of the mesh together with the FlexExtAsset pointer and a few other parameters that control the result.

The method outputs two arrays:

- SkinningWeights
- SkinningIndices

Skinned animations are quite alike. The indices and weights point at the cluster that is connected with a particle. Every position/vertex has exactly 4 weights and 4 indices. Using this data together with the data of our clusters, I could now start with rendering the body.

6.5 Rendering the body

To start with, I tried to execute the transformation of the skinning on the CPU every frame. Although this worked like intended, it quickly became clear that doing all these calculations on the GPU would be far more efficient. Before coding and simply sending all the data through as a shader variable, I thought of a good way to get the data to the GPU without wasting or duplicating any data from the MeshFilter.

What I mean with 'saving data' is that every body has its own cluster/skinning system while the mesh data is exactly the same. It would be a waste to duplicate the MeshFilter's data for every body. The best way to go around this, is to have two separate vertex buffers, one for the mesh data and one for the skinning data. This way, I can use the same MeshFilter for multiple bodies while having separate data for the skinning. The InputLayout finally looked like this:

```
//Per meshfilter
{ "POSITION", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 0, D3D11_INPUT_PER_VERTEX_DATA, 0 },
{ "NORMAL", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 12, D3D11_INPUT_PER_VERTEX_DATA, 0 },
{ "TEXCOORD", 0, DXGI_FORMAT_R32G32_FLOAT, 0, 24, D3D11_INPUT_PER_VERTEX_DATA, 0 },
//Per softbody
{ "WEIGHT", 0, DXGI_FORMAT_R32G32B32A32_FLOAT, 1, 0, D3D11_INPUT_PER_VERTEX_DATA, 0 },
{ "CLUSTER", 0, DXGI_FORMAT_R32G32B32A32_SINT, 1, 16, D3D11_INPUT_PER_VERTEX_DATA, 0 },
```

Only having the weight and the cluster to calculate the transformations is not enough. Doing those translations requires the position, rotation and restposes of each cluster.

Since in a shader, I can't change the size of an array during runtime, I set the maximum amount of clusters to 200. If there is a body that exceeds this amount, it will throw an error.

```
float3 gRigidRestposes[200];
float3 gRigidTranslations[200];
float4 gRigidRotations[200];
```

Now everything is sent to the GPU, I could finally get to the actual transformation of each vertex so that it can be rendered.

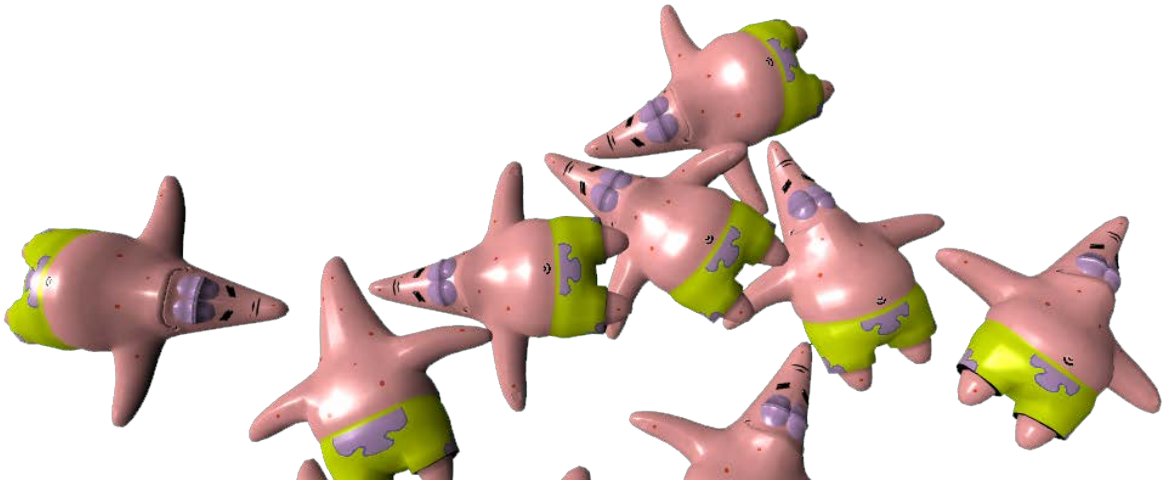
As said, every vertex has 4 weights and the sum of these weights is always 1. To transform a soft body, we need to transform the position and the normal of each vertex according to the position and rotation of every cluster connected to that vertex. The vertex shader can be read below. As for the pixel shader, it does not affect the mesh itself so the shading can be done any way you like. For simplicity, I rendered the body with simple blinn-phong shading.

```
float3 softNormal = float3(0,0,0);
float3 softPos = float3(0,0,0);
for(int w = 0; w < 4; ++w)
{
	int rigidIndex = input.cluster[w];
		float3 localPos = input.pos - gRigidRestposes[input.cluster[w]];

	float3 skinnedPos = gRigidTranslations[rigidIndex] +
	rotate(gRigidRotations[rigidIndex], localPos);
	float3 skinnedNormal = rotate(gRigidRotations[rigidIndex],
	normalize(input.normal));
	softPos += skinnedPos * input.weight[w];
	softNormal += skinnedNormal * input.weight[w];
}
output.wPos = softPos;
output.pos = mul(float4(output.wPos, 1), gViewProj);
output.normal = normalize(softNormal);
```

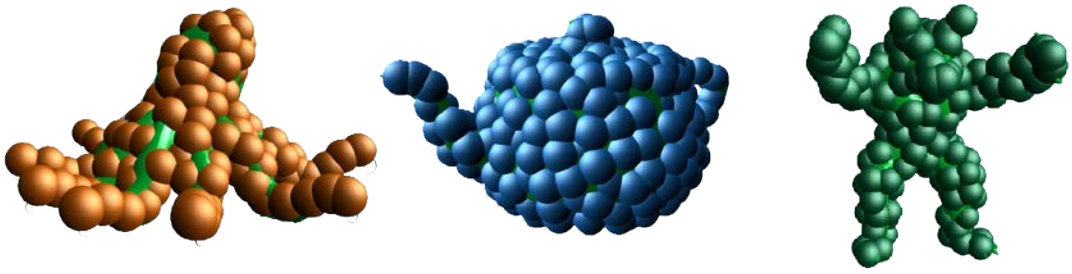
6.6 Final result

After the whole process of creating the body, uploading the data, creating the skinning and rendering it, we can finally see a satisfying result.



Knowing how to create a soft body, makes it really simple to create rigid bodies. A rigid body is basically a soft body but with only one cluster. This means we only need to send one translation, rotation and restpose to the GPU to transform all the vertices. Since the helper function 'flexExtCreateSoftFromMesh' creates multiple clusters, we can't use this function. This is where flexExtCreateRigidFromMesh() comes in. As the other function, it outputs a FlexExtAsset pointer but only with one cluster. We don't need any skinning data since everything is bound to one rigid.

Because the bodies are created directly from a mesh filter, the possibilities are endless. Any mesh can be loaded and converted to a soft body. Below are a few meshes I converted to a soft body.



At the start of my research, I had the chance to try the Nvidia VR funhouse game and noticed you had the ability to throw an octopus at a bullseye and it would stick to the board when aimed right. I looked into this and couldn't find any reference to particles sticking to collision meshes. The solution was eventually quite simple and easily overlooked. There is an adhesion parameter in the FlexParams that makes particles more 'sticky'. I started experimenting with this but came to the conclusion that this would never work perfectly. I needed a specific object to be sticky so I could throw something at it so you would think that adding a second solver for this would work. Unfortunately, that doesn't work since there is no collision between particles of different solvers. This meant that if you need an object to be sticky, everything that needs to interact with that object will have the exact same parameters.

7.0 Triangle mesh collision

Besides just having soft bodies working, I needed something that the soft body can interact and collide with apart from other soft bodies or rigid bodies. Soft bodies and rigid bodies are great for dynamic purposes but take a lot of resources This is where triangle collision meshes come in. These collision meshes can be created and changed during runtime using the FleX framework. First, you have to create a triangle mesh and with that triangle mesh you can create a collision geometry.

7.1 Creating the triangle mesh

Triangle meshes in FleX are created using the flexCreateTriangleMesh() method. This method returns a pointer to a piece of data that will contain all the information of the triangle mesh. This object does not contain any data yet so we have to provide it to the object using the flexUpdateTriangleMesh() method. This method takes in a few parameters.

[in]	mesh	A triangle mesh created with flexCreateTriangleMesh()
[in]	vertices	Pointer to an array of float3 vertex positions
[in]	indices	Pointer to an array of triangle indices, should be length numTriangles*3
[in]	numVertices	The number of vertices in the vertices array
[in]	numTriangles	The number of triangles in the mesh
[in]	lower	A pointer to a float3 vector holding the lower spatial bounds of the mesh
[in]	upper	A pointer to a float3 vector holding the upper spatial bounds of the mesh
[in]	source	The memory space the transfers to the GPU will occur from

The function requires the lower and upper bounds of the mesh in local space. This can be obtained by finding the two extreme points of the mesh. I used the BoudingBox class from the DirectXCollision.h header that created a boundingbox by providing the method the vertices of a mesh. With a bounding box, I could easily find the extremes by adding the extents of the bounding box to the center for the upper bounds and subtracting the extents from the center for the lower bounds. With this data, you can create the triangle mesh. At this moment, I had a FlexTriangleMesh object which is a data representation of a triangle mesh on the physics solver.

```
flexUpdateTriangleMesh(
	pMesh,
	(float*)pMeshFilter->GetVertexData("POSITION").pData,
	(int*)pMeshFilter->GetVertexData("INDEX").pData,
	pMeshFilter->VertexCount(),
	pMeshFilter->IndexCount() / 3,
	(float*)&min,
	(float*)&max,
	m_pFlexSystem->MemoryType
);
```

7.2 Creating and updating the collision mesh

With the triangle mesh object created earlier, I could now create a collision geometry object that can be stored and be taken into account by the solver. For a collision geometry to work, the solver requires quite an amount of data. In the end, all of the shape data will be sent to the solver using the flexSetShapes() method and it has 13 parameters. Below, the signature of the method.

```
void flexSetShapes(
       FlexSolver*, FlexCollisionGeometry*, int numGeometryEntries, float*
       shapeAabbMins, float* shapeAabbMaxs, int* shapeOffsets, float*
       shapePositions, float* shapeRotations, float* shapePrevPositions, float*
       shapePrevRotations, int* shapeFlags, int numShapes, FlexMemory
);
```

To start with, the method requires a pointer to an array of FlexCollisionGeometry objects. FlexCollisionGeometry is a class that can accommodate data for a triangle, sphere, capsule, plane or SDF mesh. In the case of a triangle mesh, the data structure has just two variables, the scale of the mesh and the triangle mesh itself (which is the triangle mesh object created before).

Next up is an array of lower and upper bounds of the meshes. These can be obtained by using the method used before (see BoundingBox). Another advantage of using the bounding box us having the ability to transform it using the world matrix of the body.

After this comes the translations and rotations. Note that the method requires you to provide the transformation of the current frame and that one of the previous frame. This means that every frame, the current transformation gets stored in the ‘previous’ one and gets updated with a new one (that one from the transform component in my case). This allows you to move and rotate the mesh during runtime.

The last parameter except for the amount of shapes and the FlexMemory, is an array of shape flags. Since the FlexCollisionGeometry has the flexibilty to be different kinds of meshes, the flag tells the solver what kind of shape the user intends to use. A flag can be created with the helper method flexMakeShapeFlags(). It takes in an extra bool telling if the shape will be dynamic or not. In the case of a triangle mesh, the method looks like this:

```
int flag = flexMakeShapeFlags(eFlexShapeTriangleMesh, false)
```

At this point, all the data is obtained and can be passed to the solver and used in the simulation.

Creating other sorts of collision geometry like capsules, planes or spheres is more trivial since it only requires you to provide the structure with data like radius, height, length, … Besides this, there is no difference in method to create a collision mesh.

8.0 Runtime parameter controlling

It is possible to load in any mesh from a file, voxelize it, skin it and eventually render it with the given parameters. During runtime, I am able to upload changes to the FlexParams to influence the behavior of the individual particles. To be able to control these, I needed a simple user interface with sliders, checkboxes, buttons,…

There is a free framework called ImGui (https://github.com/ocornut/imgui). ImGui is a graphical user interface library for C++ that supports everything I needed. The framework has a static class that creates command lists that can be retrieved by the user to send to the graphics context, in my case the DirectX device context, to render each frame.

Integrating the framework requires some work because you have to create buffers, create the shader and transfer the command lists of the framework to the graphics context yourself. For this I created a class ImgUIDrawer that handles the initializing, resource managing, drawing and updating of the UI while still using the static methods of the framework to create the layout of the UI.

Once this is all done, using the framework is extremely simple. Below is the result of a piece of code that creates UI elements to control the FlexParams.

```
ImGui::Begin("Nvidia FleX");

ImGui::Separator();
ImGui::SliderInt("Substeps", &m_pFlexSystem->Substeps, 0, 10);
ImGui::SliderInt("Iterations", &m_pFlexSystem->Params.mNumIterations, 0, 10);
ImGui::Separator();
ImGui::SliderFloat("Gravity X", &m_pFlexSystem->Params.mGravity[0], -10, 10);
ImGui::SliderFloat("Gravity Y", &m_pFlexSystem->Params.mGravity[1], -10, 10);
ImGui::SliderFloat("Gravity Z", &m_pFlexSystem->Params.mGravity[2], -10, 10);
ImGui::Separator();
ImGui::SliderFloat("Radius", &m_pFlexSystem->Params.mRadius, 0, 4);
ImGui::SliderFloat("Soid Radius", &m_pFlexSystem->Params.mSolidRestDistance, 0, 4);
ImGui::SliderFloat("Fluid Radius", &m_pFlexSystem->Params.mFluidRestDistance, 0, 4);
ImGui::Separator();
ImGui::SliderFloat("Dynamic Friction", &m_pFlexSystem->Params.mDynamicFriction, 0, 1);
ImGui::SliderFloat("Static Friction", &m_pFlexSystem->Params.mStaticFriction, 0, 1);
ImGui::SliderFloat("Particle Friction", &m_pFlexSystem->Params.mParticleFriction, 0, 1);
ImGui::SliderFloat("Restitution", &m_pFlexSystem->Params.mRestitution, 0, 1);
ImGui::SliderFloat("Damping", &m_pFlexSystem->Params.mDamping, 0, 1);
ImGui::SliderFloat("Plastic Creep", &m_pFlexSystem->Params.mPlasticCreep, 0, 1);

ImGui::Separator();
ImGui::Text("Particle Count: %i", m_pFlexSystem->Positions.size());
ImGui::Text("Rigid Count: %i", m_pFlexSystem->RigidTranslations.size());
ImGui::Text("Collision Meshes: %i", m_pFlexSystem->ShapeGeometry.size());

ImGui::PlotLines("Frame times", GameTimer::GetFrameTimes().data() ,
    GameTimer::GetFrameTimes().size(), 1, nullptr, 0.0f, 0.05f, ImVec2(0, 80));

ImGui::Checkbox("Simulate", &m_FlexUpdate);
if (ImGui::Button("Restart scene"))
    GameManager::GetInstance()->LoadScene(new SoftBodyScene());
if (ImGui::Button("Toggle debugging"))
    m_pFlexDebugRenderer->ToggleDebugging();
```

9.0 Fluids

Besides soft bodies, FleX has many more possibilities. Another big application is fluid simulations. Because time allowed me to, I did some research in that topic as well. As with every FleX feature, the basic building blocks always stay the same. Fluid particles don't need any voxelization or skinning but merely has different FlexParams that makes the particles behave more fluid-like. Together with these specific parameters for fluids, a special rendering technique makes the particles look like fluids.

9.1 Parameters

As said, the FlexParams play a big role in the behavior of the particles, especially for fluid simulations. The FlexParams has a set of parameter I hadn't used before for soft bodies since they all relate to fluids.

float	mFluidRestDistance	The distance fluid particles are spaced at the rest density. Range [0, radius]
bool	mFluid	Particles with phase 0 are considered fluid particles and interact using the position based fluids method.
float	mCohesion	Control how strongly particles hold each other together, default: 0.025, Range [0.0, +inf].
float	mViscosity	Smoothes particle velocities using XSPH (Smoothed particles hydrodynamics) viscosity.
float	mFreeSurfaceDrag	Drag force applied to boundary fluid particles.
float	mBuoyancy	Gravity is scaled by this value for fluid particles.

With these parameters in mind, I looked at the demo application and started experimenting with different combinations ranging from viscous behavior to water-like behavior.

9.2 Rendering

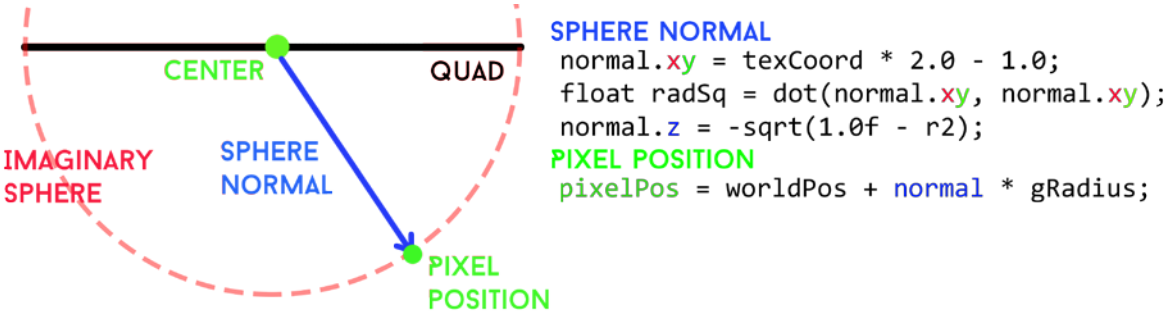
In the case of fluids, creating the particles and simulating them is easy but the idea behind rendering is more complex. There are multiple ways to render the particles and one might say marching cubes is the way to go but this is very performance heavy in a real-time environment and needs predefined bounds to work.

Another approach is the screen-space fluids algorithm (SSF) which is inspired by the "Screen Space Meshes" paper by Simon Green (2010). This algorithm works entirely in screen space and does not involve any meshes. The process consists out of these steps:

1. Create billboarded quads on the position of each particle using a geometry shader
2. Output the depth of a sphere to a separate rendertarget
3. Blur the depth image
4. Recreate the normals from the depth image
5. Render the surface using the calculated normals

I've put some time into making a practical implementation of this but stumbled across quite some complications when trying to reconstruct the normals from the depth map. I rendered a linear depth map from the quads that immitated spheres by translating

the center of the quad by the normals of a sphere of that pixel. Below is an image that explains this idea on a quad seen from top view.



In a second shader I tried to retrieve the normals of the depth map but failed to achieve a good result. This was caused by imprecisions of the depth-to-normal algorithm and the imprecision of the depth texture. The result contained too many artifacts to achieve an acceptable result.

I decided to let the topic be and continue researching soft bodies and the rest of the FleX framework. This is definitely something I will look into in the future.

10.0 Notes

10.1 Nvidia FleX and Nvidia PhysX

Nvidia FleX is a framework completely separated from any other framework from Nvidia like PhysX. This introduces quite some complications since there is absolutely no interaction between both of them. For example a rigidbody from PhysX will not interact with a collision mesh from FleX meaning that if you would like to combine these frameworks, you would need to do double the work. As stated by a user:

FleX is not designed to build gameplay affecting physics, as it lacks functionality such as trigger events, contact callbacks, ray-casting, serialization, etc. For this reason it is recommended to use FleX in conjunction with a traditional rigid-body physics engine, such as PhysX SDK.

It makes sense that there are no interactions between the two frameworks but it definitely decreases the usage of Nvidia FleX since it introduces all these complications.

Miles Macklin, the lead developer of FleX stated that in the future, the new release of Nvidia PhysX (version 3.4), Nvidia FleX will be integrated into PhysX and manual mirroring will no longer be required since the systems will share the same collision detection pipeline.

10.2 Other notes

The demo application from Nvidia is the best source for reference but still lacked some detail that still has to be figured out. A great example of this is the FlexContainer class that could potentially improve the performance of the application by lowering the overhead of downloading and uploading the particle data of the GPU.

```
FLEX_API FlexExtContainer* flexExtCreateContainer ( FlexSolver * solver,
                                                    int          maxParticles
                                                  )
```

Creates a wrapper object around a Flex solver that can hold assets / instances, the container manages sending and retrieving partical data from the solver

Parameters

[in]	**solver**	The solver to wrap
[in]	**maxParticles**	The maximum number of particles to manage

Returns

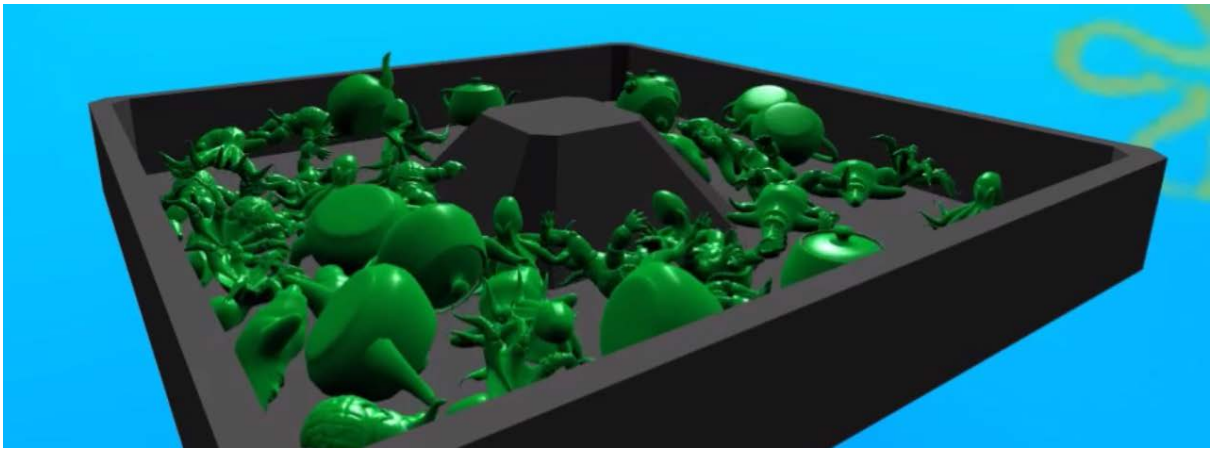
A pointer to the new container

It would be interesting to look into this container structure as it might be a good way to contain all the data of the particles in a well-designed structure.

Another example is the memory location of the particle data. As I mentioned in the part where I explained the phases, when uploading or downloading data to the solver, the last parameter is the memory source. This is an enumeration eFlexMemory that can be either eFlexMemoryHost or eFlexMemoryDevice. This basically means that the data can be stored either on the CPU or the GPU. I have done some research in to

this but at this point I can not use the GPU for data storage since this causes errors. The problem is that neither the demo or the documentation contains information that documents this enumeration and how to use it.

All these optimizations could make me able to take a step further in the amount of particles I am able to create and the complexity of the scene. As a final test, I wanted to test the limits of the framework on my device by checking how many softbodies I could simulate at the same time. I was quite surprised to see that it could handle about 50 soft bodies which are about 14 000 particles all together.



10.3 Reflection

After a few months of researching the Nvidia FleX framework, I learnt quite a lot. Not only how to work with the framework itself but also how to start a research without having any knowledge in advance. It was though at first to start comprehending the demo application provided in the SDK and learn from it and eventually apply it.

I have decided to create all the applications and demos during this research in my own engine which caused some other complications as well. The upside of this was that I had a great overview and sense of control over what I was working with. Creating this engine myself helped me a lot since it gave me a better understanding of graphics programming in general which in its turn helped me to understand the rendering techniques of visualizing a soft body.

I understand why so few games make use of the framework. On its own, the framework is too barebone and does not provide enough control to be properly used in games. The performance of the simulations are quite promising though and I am certain that in the future update of PhysX, Nvidia FleX will be used more often.

With this research I achieved what I intended to achieve, that is being able to simulate, render and dynamically control a soft body. I will definitely continue looking into the framework and try to understand the other applications of Nvidia FleX.

11.0 Acknowledgements

I was guided on this project by Thomas Goussaert, lecturer at Digital Arts & Entertainment, Kortrijk. I am thankful for his insight, ideas and feedback throughout the researching process.

The engine used for this research is inspired by the Overlord Engine, written by Thomas Goussaert.

12.0 References

Miles Macklin, Matthias Müller (2014), Unified Particle Systems
http://blog.mmacklin.com/flex/
Quan Chen (2015) , Making your game fully interactive by Nvidia FleX
http://developer.download.nvidia.com/assets/gameworks/downloads/regular/events/cgdc15/Making%20Your%20Game%20Fully%20Interactive%20by%20NVIDIA%20FleX-ENG.pdf
Simon Green (2010), Screen space fluid rendering in games
http://developer.download.nvidia.com/presentations/2010/gdc/Direct3D_Effects.pdf
The Nvidia FleX documentation and demo application
http://docs.nvidia.com/gameworks/content/gameworkslibrary/physx/flex/
The Unreal Engine source code on GitHub
https://github.com/NvPhysX/UnrealEngine
The official framework page
https://developer.nvidia.com/flex
The PhysX wiki
http://physxinfo.com/wiki/PhysX_FleX
Armen Barsegyan (2016), NVIDIA PhysX FleX and other fluid solvers for high-quality fluid simulation
http://cgicoffee.com/blog/2016/11/nvidia-physx-flex-fluid-simulation-and-other-solvers

All images created by myself