AUGMENTED CLOTH FITTING WITH REAL TIME PHYSICS SIMULATION USING TIME-OF-FLIGHT CAMERAS

A THESIS

SUBMITTED TO THE DEPARTMENT OF COMPUTER ENGINEERING

AND THE GRADUATE SCHOOL OF ENGINEERING AND SCIENCE

OF BİLKENT UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

MASTER OF SCIENCE

By

Umut Gültepe

December, 2012

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Assoc. Prof. Dr. Uğur Güdükbay (Supervisor)

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

(Co-supervisor)

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Approved for the Graduate School of Engineering and Science:

Prof. Dr. Levent Onural
Director of the Graduate School

ABSTRACT

AUGMENTED CLOTH FITTING WITH REAL TIME PHYSICS SIMULATION USING TIME-OF-FLIGHT CAMERAS

Umut Gültepe
M.S. in Computer Engineering
Supervisors: Assoc. Prof. Dr. Uğur Güdükbay

December, 2012

This study surveys and proposes a method for an augmented cloth fitting with real time physics simulation. Augmented reality is an evolving field in computer science, finding many uses in entertainment and advertising. With the advances in cloth simulation and time-of-flight cameras, augmented cloth fitting in real-time is developed , to be used in textile industry in both design and sale stages. Delay in cloth fitting due to processing time is the main challenge in this research. Human body is identified, articulated and tracked with a time-of-flight camera. Depending on the size and position of body limbs, a virtually simulated cloth is fitted in real time on the subject. Delay is reduced with GPU computing for cloth simulation and collision detection.

Keywords: cloth simluation, computer vision, natural interaction, virtual fitting room, kinect, depth sensor.

ÖZET

UÇUŞ ZAMANI KAMERALARI KULLANAN GERÇEK ZAMANLI FİZİK SİMÜLASYONLU ARTIRILMIŞ KIYAFET KABİNİ

Umut Gültepe
Bilgisayar Mühendisliği, Yüksek Lisans
Tez Yöneticileri: Doç. Dr. Uğur Güdükbay

Aralık, 2012

Bu çalışma fizik simülasyonlu bir artırılmış kıyafet giydirme için bir metot önermekte ve incelemektedir. Artırılmış gerçeklik bilgisayar biliminde gelişen bir alandır, eğlence ve reklam sektörlerinde geniş yer bulmaktadırlar. Kumaş simülasyonu ve uçuş-zamanı kameralarının geliştirilmesi ile, gerçek zamanlı artırılmış kıyafet giydirimi tekstil endüstrisinde tasarım ve satış aşamalarında kullanılmak üzere geliştirilmiştir. şleme zamanı sebebiyle kıyafet giydirilmesindeki gecikme bu araştırmadaki en büyük zorluktur. nsan vücudu bir uçuş-zamanı kamerası ile tanımlanmakta, bölünmekte ve takip edilmektedir. Vücut parçalarının boyutlarına ve pozisyonlarına göre simüle edilen bir sanal kıyafet kullanıcının üstüne yerleştirilmektedir. Gecikme zamanı kıyafet simülasyonunu ve çarpışma takibini GPU üzerinde yaparak azaltılmaktadır.

Anahtar sözcükler: kıyafet simülasyonu, bilgisayarla görü, doğal etkileşim, sanal giyinme kabini, kinect, derinlik sensörü.

Acknowledgement

I am deeply grateful to my supervisor Assoc. Prof. Dr. Uğur Güdükbay, who guided and assisted me with his invaluable suggestions in all stages of this study. I also chose this area of study by inspiring from his deep knowledge over this subject.

I am very grateful to Computer Engineering Department of Bilkent University for providing me scholarship for my MS study.

I would like to thank Scientific and Technical Research Council of Turkey (TÜBİTAK) and the Turkish Ministry of Industry and Technology for their financial support for this study and MS thesis.

to my mother, father and my brother...

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Overall Framework-OGRE

As a programmer, I embrace the do not repeat yourself mindset and extreme programming methodology. Furthermore, since my study required utilization of many different techniques in different fields in Computer Science, I searched for various 3rd party SDKs to save me from writing ground-level code. These two beliefs led me to search for the rendering engine best suited for my needs:

1. Must be code oriented rather than designer oriented.
2. Must be able to utilize both DirectX and OpenGL (for compatibility reasons).
3. Must take care of mundane and routine programming such as the rendering pipeline, input handling,
4. Must be able to integrate easily with 3rd party libraries.
5. Must be stable and mature.
6. Must have an associated 3-D designing program which can be used to easily produce content and load into the program.
7. Must have accurate and extensive documentation.

Other than these, automatic material rendering, skeletal animation support were considered as bonuses. After experimenting with Unity, UDK and native OpenGL programming, I eventually decided on Object-Oriented Rendering Engine (will be referred as OGRE in the rest of the paper) for it complies with my requirements the best.

1.1 The Features

Unlike the name suggests, OGRE is more than just a rendering engine. Among all the features, the ones I utilized are as following [7]

- Render State Management
- Spatial Culling and Transparency Handling
- Material Rendering
  - Easy material and shader management, custom shader support.
  - Multitexture and Multipass blending
  - Lighting shader and different shadow rendering techniques
  - Material Level Of Detail Support
  - Supports a variety of image formats, volumetric textures and DXT textures.
  - Render-To-Texture (Frame Rendering Buffer) support
- Meshes
  - Native mesh format which can be exported from Blender Designer.
  - Level Of Detail Support
  - Skeletal animation feature, can be used with models exported from Blender.
    - Multiple-bone weighted skinning

    * Hardware Acceleration.

- Easy scene management
- Easy camera and input management.
- Easy integration with 3rd party libraries due to code-based nature.
- Overlay feature which enables easy information tracking about the feature.

These features led me to choose OGRE as my base framework.

1.2 High Level Overview

The class diagram in Figure 1.1 shows the Object-Oriented core of the OGRE [10]:

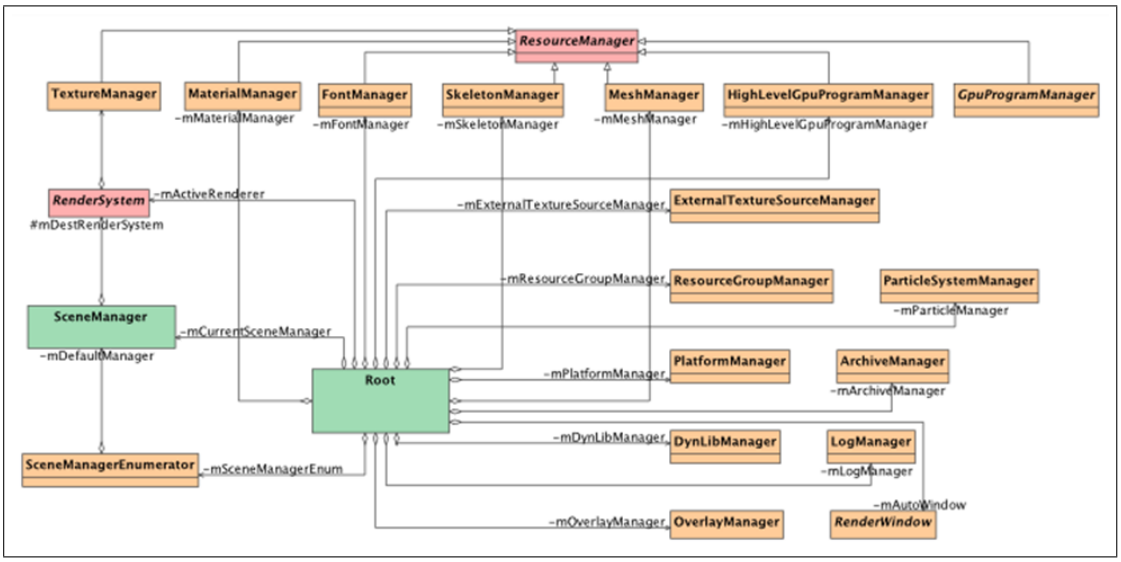

Figure 1.1: OGRE High Level Overview

1.2.1 The Root object

The root object is the entry point and core of the framework.

- It is created first and destroyed last in the application lifecycle.
- It configures the system, delivers pointers to the managers for various resources.
- Provides automatic rendering cycle, continued until an interrupt from FrameListener objects.

1.2.2 The RenderSystem Object

Render system is an abstract class to define the underlying 3D API (either Direct 3D or OpenGL). This class is not accessed and modified by the application programmer.

1.2.3 The SceneManager Object

SceneManager is the most used object by the application programmer, as it is in charge of the contents in the scene to be rendered.

- It is used to create, destroy and update the objects.
- It sends the scene to the RenderSystem behind the curtains for rendering.
- Multiple SceneManagers can be used to create other visual resources (ex. RenderToTexture environment)

1.2.4 Resource Manager

ReosurceManager object is an abstract class, used to create, keep and dispatch a type of resource it is associated with.

- The associated type is defined by the class inheriting the ResourceManager, such as MaterialManager.

- There is always only one instance of every child of ReosurceManager in an application.
- Resource Managers search the pre-defined locations of the filesystem and automatically indexes the resources available, ready to be loaded upon demand.

1.2.5 Entities, Meshes and Materials and Overlays

Entities are the instances of movable objects in the scene. They are based on meshes, which define the geometric and material properties of the entity. Materials contain information about what color the final pixel in the rendering should be.

- Entities are attached to scene nodes for moving and rendering. Scene nodes can be nested, which greatly simplifies the process of rendering complex scenes.
- Meshes consist of submeshes, which can have different material associations. Therefore, a mesh can be composed of various parts with various materials.
- Materials are defined either in run-time or in .material scripts, with detailed information. They also support custom shaders.
- Mesh files can be created and saved with manual objects, or exported through designer programs such as Blender. The .mesh files are in binary format.

Overlays are used to create panels for control and HUD which are rendered above the scene. They are 2D elements are placed either by screen proportion or pixel size and rendered orthographically, last in the rendering pipeline by default (this can be overridden).

1.2.6 The render cycle

The self-explanatory render cycle of OGRE is given in Figure 1.2 [10].

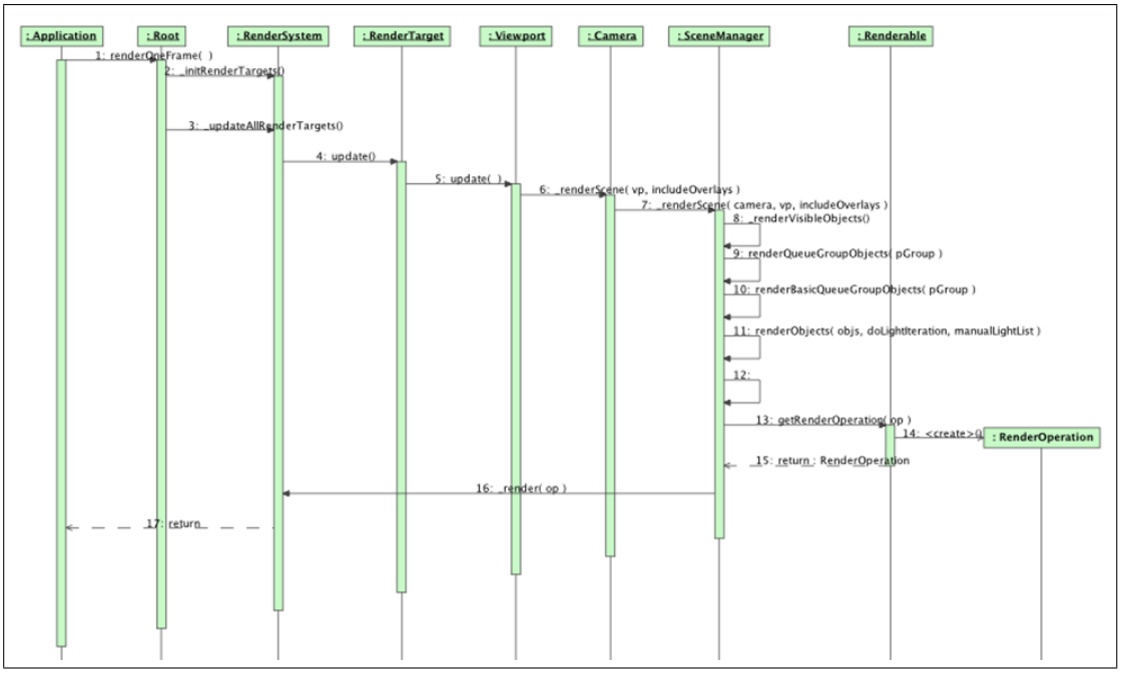

Figure 1.2: OGRE Render Cycle

Chapter 2

User Tracking

2.1 Hardware

User tracking has always been both a challenge and a valued feature in Image Processing. Until the availability of Time-Of-Flight cameras, user tracking was dependent on RGB cameras. Although RGB cameras are sufficient for user tracking, they are proven to be harder to use for body articulation and joint estimation, mostly because of the complexity of human body, self occlusions and the difficulties of body segmentation based on pixel colors alone. Researchers started with still images, than experimented with image sequences from multiple cameras. Most common technique was to extract the body silhouette from the image and construct a body shape with silhouettes from multiple calibrated cameras-Shape From Sihouette (SFS) [3]. SFS algorithms evolved from spatial to temporal tracking and their accuracy improved even more [3].

However, RGB based accurate user articulation techniques required many cameras-a financial problem. After the need for extensive imaging hardware, the processes required to calibrate the cameras initially, extract and combine the silhouettes, build a shape and articulate the result. These processes require very complex algorithms, many man-hours to implement and very powerful computing infrastructure. All these requirements made shape-from-silhouette algorithms

hard to utilize for me.

Instead of RGB imaging, I searched for an alternate type of device which can capture the depth of the field in front of it. Although there are a variety of such devices, from sonars to Laser Scanners, the one most appropriate for user tracking was Time-Of-Flight cameras. They have significant advantages over Stereo-Vision and Laser Tracking in simplicity, are fast (up to 100 fps) and accurate in distance [5].

Choosing the most accurate Time-Of-Flight camera was easy, as Microsoft Kinect was not only the cheapest and the most available of them all, it was also the most powerful and had an extensive developer community. I utilized the Kinect for XBOX rather than Kinect for Windows in this study, due to its distance and performance characteristics.

2.2 Software

The user tracking process with Kinect was much simpler compared to SFS techniques, due to the Shape being available mostly with the depth field output. However, proper articulation still required lots of different algorithms and time.

In order to speed up the user tracking and articulation, I utilized the OpenNI framework along with PrimeSense NITE package and SensorKinect drivers. OpenNI provides the framework for capturing and utilizing the various types of streams from Natural Interaction devices, also provides abstract modules for accessing complex functionalities, such as skeletal tracking [12]. PrimeSense NITE package is a software that bundles with OpenNI, and provides skeletal tracking, hand tracking and other functionalities which can be accessed via OpenNI framework [13].

With the integration of these modules to my application, I was able to acquire the joint positions and orientations from the depth sensor with almost no effort except the integration itself. With the current hardware/software configuration, I

acquire 15 joint positions and orientations at 30 fps, which is enough to reproduce the movement of the user on the virtual model. Other than the joint information, I also acquire the depth and image streams from the depth camera. I will be using the image stream to present the results I have obtained, comparing the user movement with the simulated environment.

Chapter 3

Hand Tracking

3.1 OpenCV

Although OpenNI/NITE provided me the joint information, useful functions for a smooth user interface, such as hand state recognition or hand swipe filters were omitted in the open source natural interaction libraries I use. In order to simulate mouse-clicking behavior, I decided to track the hands of the user to be used as cursors, notice open/close hand gestures for clicking events. In order to find implementations of the algorithms in my proposed hand-track solution, I researched various libraries which came with functions implementing such algorithms. I chose to work with OpenCV, due to its maturity, community and integration with other libraries.

OpenCV not only provides basic functions to perform complex and processor-intensive image processing functions such as, facial recognition system, gesture recognition, stereoscopic 3D and segmentation, it also has a very vast machine learning aspect, including boosting, decision three learning and many more features [2].

3.2 The Process

Utilizing such a powerful middleware as a depth sensor, I am able to perform very robust background subtraction with almost no effort skeletal body tracking is another embedded property of the software which I use, giving me a speed boost. Therefore, I implemented two different techniques: Hand state recognition and hand swipe recognition.

Hand swipe recognition is simpler compared to hand state recognition. It is just a matter of keeping track of the hands 3D position, and keeping a listener which is activated when the 3D velocity of the hand exceeds a certain number. I also performed a number of optimizations on this function to make it invariant with the size and location of the user.

Open/Close Hand recognition is harder than Hand Swipe recognition. I had to perform advanced image processing in order to determine the state of the hand successfully, at relatively low resolutions and large distances. My algorithm is shown in Figure 3.1:

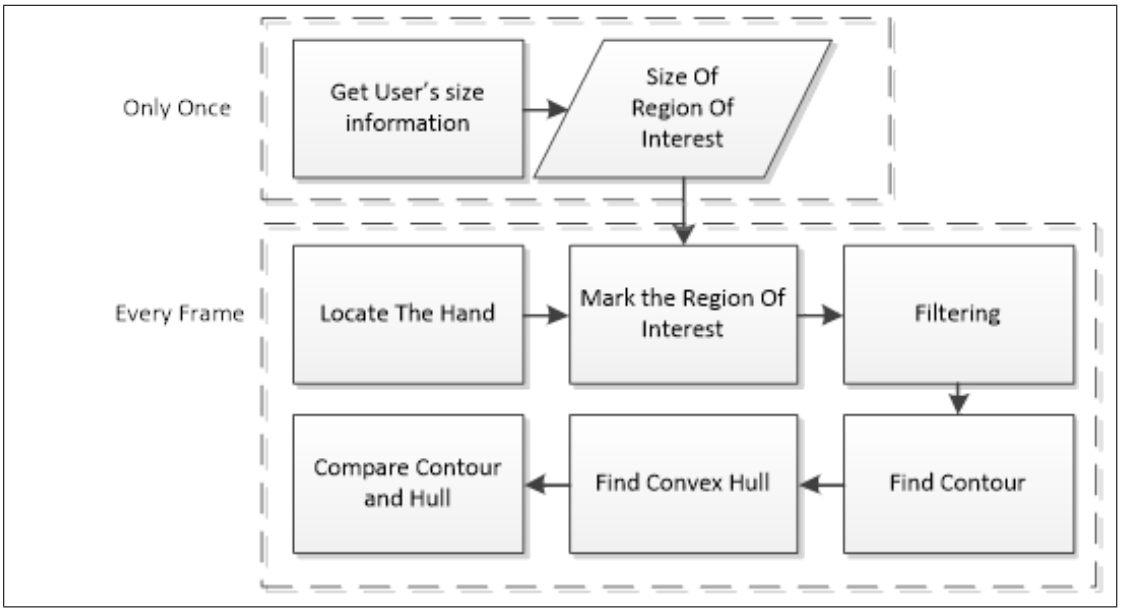

Figure 3.1: Hand Recognition Algorithm

1. The size information for the region of interest is drawn from the distance

between the users head and users neck.

2. The hand is located using skeletal body tracking.
3. The marked region is copied from the depth stream
4. Two dilation and one erosion operations are performed to smooth the hand image
5. The contour is found on the filtered image
6. The convex hull of the found contour is calculated.
7. The depth difference between the hull and the actual contour is taken as the reference for hand-state.

Figure 3.2: Hand Images and Contours from Depth Stream

The test run results can be seen in figure Figure 3.2. The left part of the figure shows the hand in open state, whereas the hand is closed on the right part.

Ultimately, I plan to implement the algorithm described in [14], where the author claims to have achieved 96% correct results. I have not included RGB image in the process yet, neither have I implemented complex machine learning. In addition to these methods, I will record hand images for testing objectively. These topics are in my future research plans.

Chapter 4

3-D Model

There are two types of 3-D models I needed for my thesis project: A cloth mesh to be simulated and a human body to dress with the simulated cloth.

4.1 Human Avatar

Although I have experience with modeling in Blender, designing a 3-D human avatar from scratch seemed unnecessary, as it is not in my area of expertise and there are available models online. After a vast search, I decided to utilize the model I have acquired from a forum, as it was realistic in appearance, proportions and details [9]. In order to make it look more realistic, I needed to implement additional features.

4.1.1 Rigging

Animating a 3D mesh requires skinning, which is moving the vertices with respect to a bone on a skeleton. The model I acquired from internet had no skinning or material information, which I meant I had to do it myself. And even it was a pre-skinned model, I would prefer doing it on my own, in order to be able to

integrate the model easily into my software.

I performed the skinning in blender, where I used the predefined human skeleton, detailed in H-ANIM2 level with a much simplified spine (Figure 4.1).

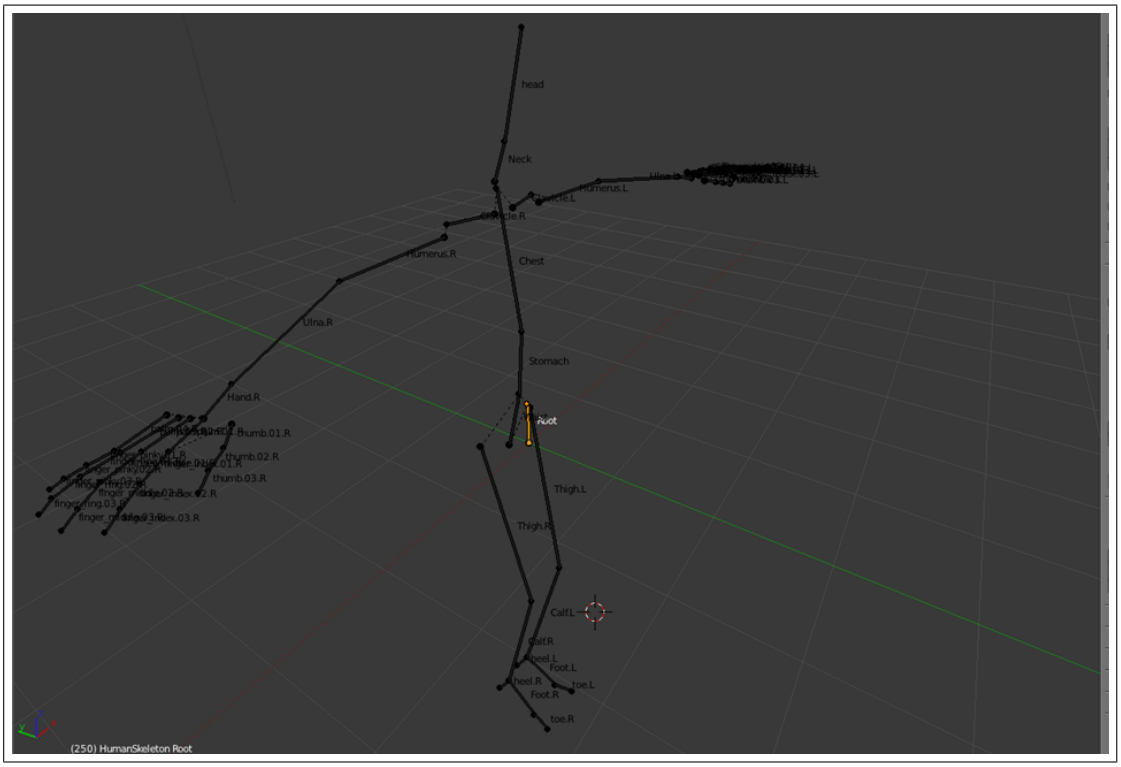

Figure 4.1: The Rigging Base Skeleton

Prior to rigging, the skeleton needed to match the given mesh in position. After modifying the initial bone positions, the vertex groups are assigned to the bones with proper weights (Figure 4.2). After assigning every vertex to the appropriate bones, there were no cracked surfaces with motions which are natural for humans.

In the end, the model was exported in OGRE .mesh format along with the skeleton it used.

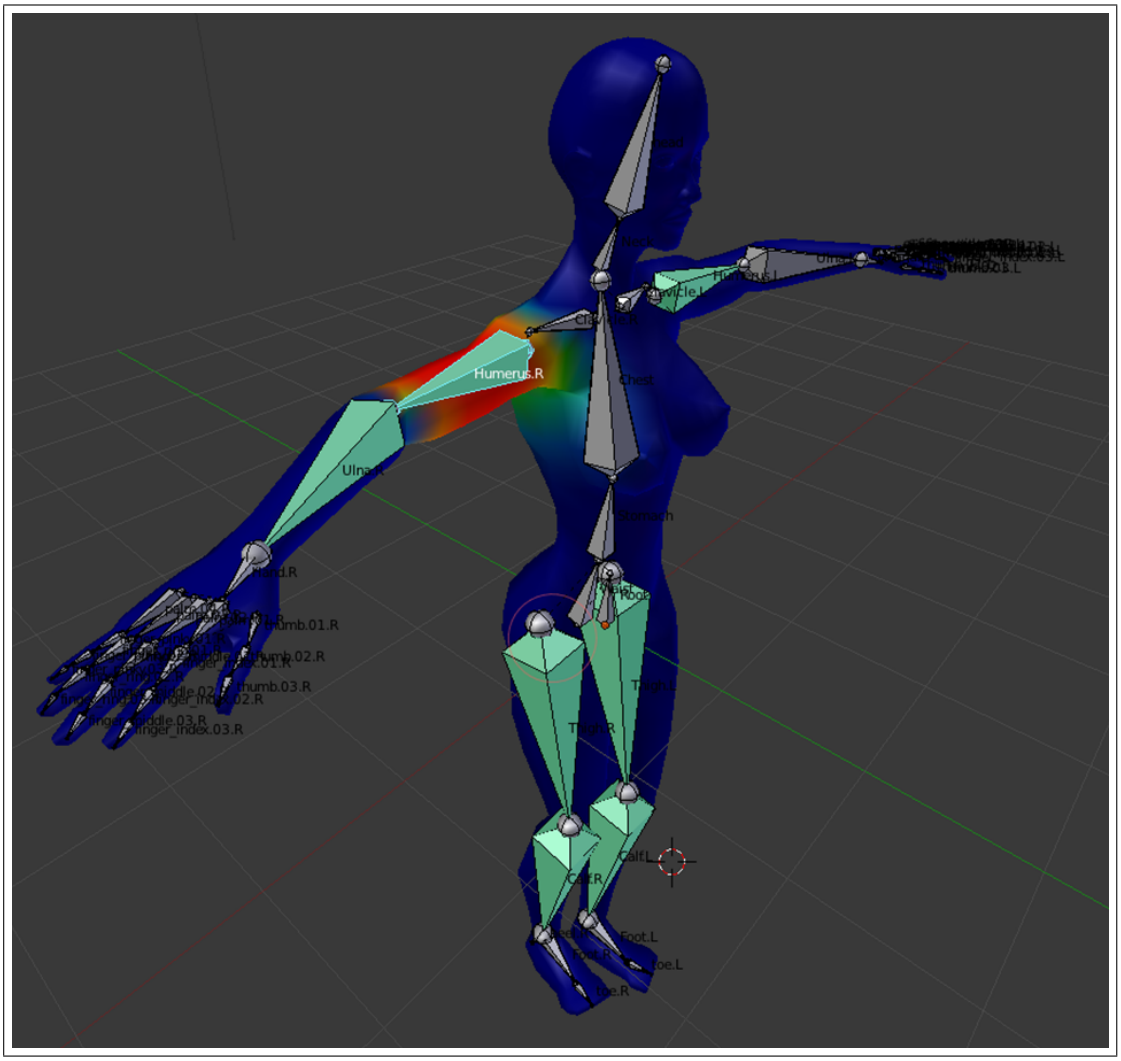

Figure 4.2: The vertex weights for the Humerus.R bone

4.1.2 Materials

The base model came with no material information. In order to improve the realism of the model, I implemented texturing/painting for various parts of the body. These parts include the general skin, eyes, nails and lips so far (Figure 4.3). I will continue material introduction to other parts I deem necessary to improve realism. I also added non-simulated hair and some accessories such as hair sticks and earrings to the model to make it look more realistic.

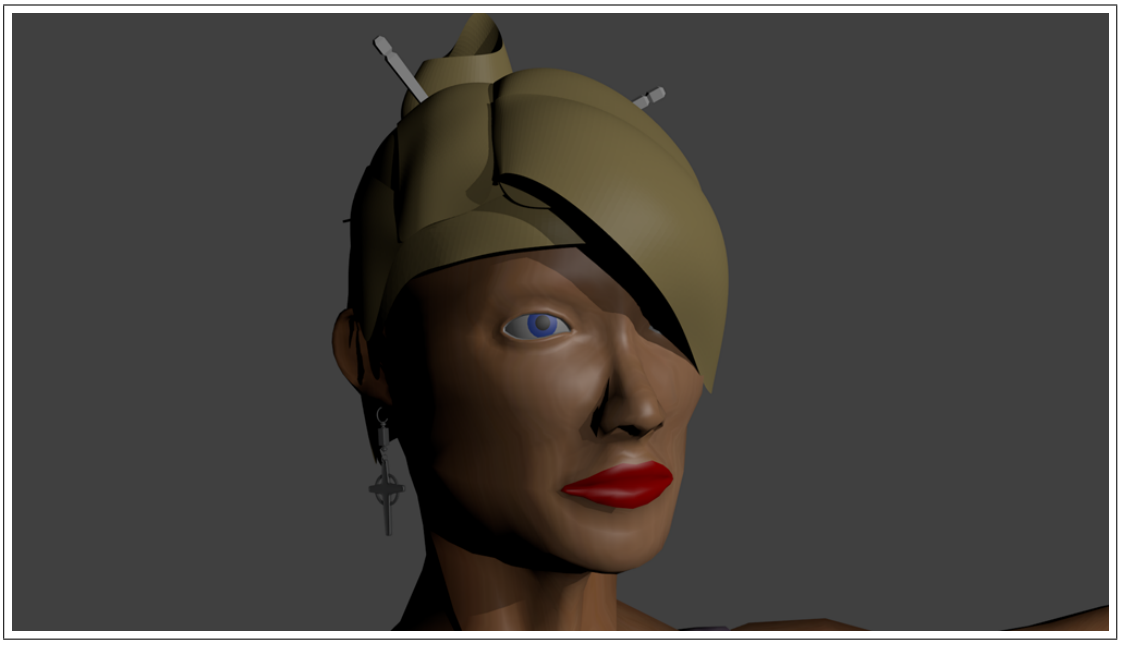

Figure 4.3: Detailed Materials on the Face

4.2 Cloth Mesh

A major part of my thesis is on accurate cloth simulation. Next to the required software and algorithms, I also needed accurately modeled cloth meshes to be simulated. After my searching and filtering through a variety of models online, I decided to work with a dress model [8]. The model comes as pure mesh, without any material information, although it was detailed well and sufficient for my research.

4.2.1 Body Positioning and Splitting the Dress Mesh

To correctly animate and simulate the dress on a human avatar, I needed them to be in proportion with each other and properly aligned. The initial positions and proportions of the human avatar and the dress mesh were set in Blender, as it was easier to perform this with a visual tool (Figure 4.4). After various attempts to simulate all the vertices on the dress mesh, I failed to achieve a realistic looking results due to two main reasons:

- The whole dress consisted of 4088 vertices. Although I was able to simulate in real-time, too many vertices resulted in the simulation algorithm to break down due to the very large number of vertices in the cloth. It shifted from a fabric structure to more of a jelly form, over stretching and tearing with its own weight.

- The friction necessary to keep the dress on the Human Avatars shoulders was not sufficient enough to keep the dress on the avatar. It kept sliding, stretching and acting unnaturally.

In order to overcome these problems, I decided to split the dress mesh into two parts, and utilize different animation techniques on them.

- The top part, which should be attached to the body, not blowing in the wind, was modeled as a static mesh, with skinning like the human avatar. It was animated with the same transformations as the human avatar, matching its position and staying on the body perfectly.

- The bottom part was the part to be simulated, affected by inertia, wind and other factors. Their attachment line is just above the waist of the human avatar, which was confirmed after various experimentations with other locations (Figure 4.4). Keeping the line too below resulted in both unnatural collisions and the cloth being too rigid. Keeping it too high brought out the original problems. With its current setting, the dress mesh is animated naturally on the virtual avatar.

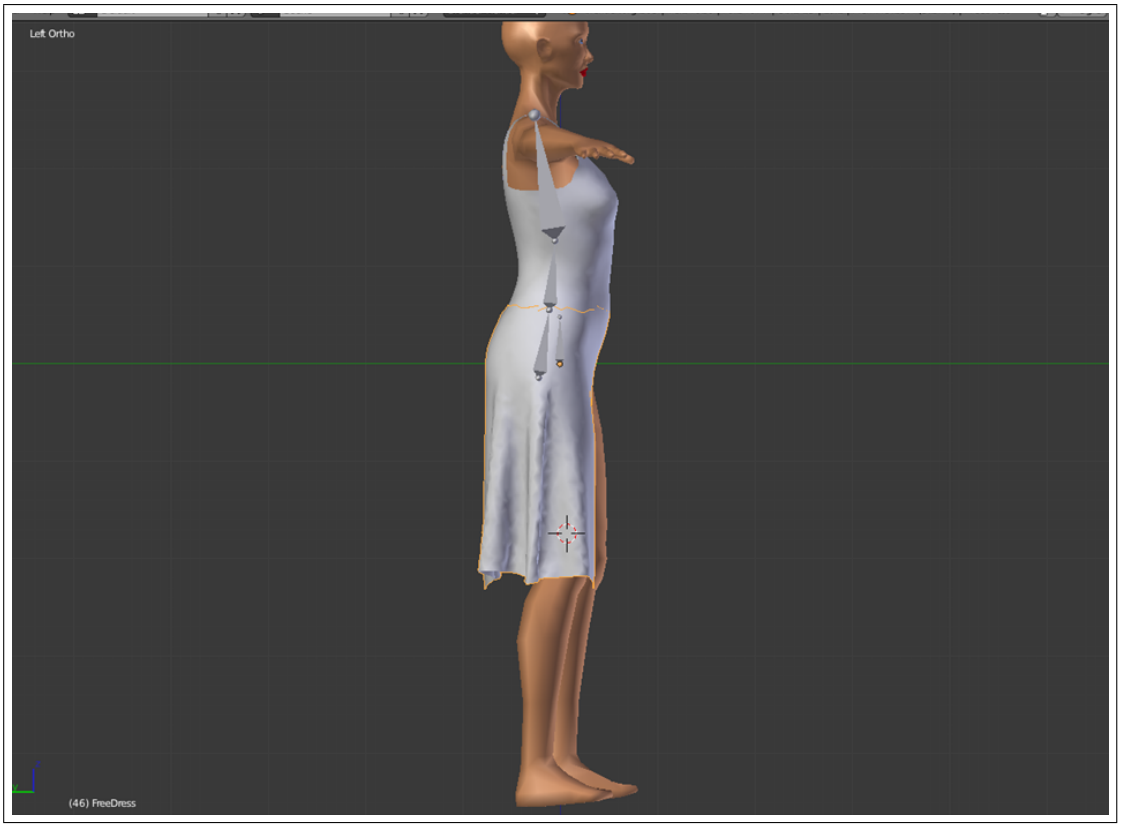

Figure 4.4: The Dress, positioned on the body, along with the upper-part skeleton. In this shot, the bottom part is highlighted in orange border.

Chapter 5

Animation

The animatable body exported by Blender is in Ogre xml mesh and skeleton format. They are converted to binary skeleton and mesh files by the OgreXML serializer, and imported natively into the software. Prior to animation, the bones are initialized to be oriented by the data from depth sensor. Afterwards, they are updated with the orientation data.

5.1 Initialization

Animatable meshes are loaded and maintained via the SkeletalMesh class I have implemented, which bridges the gap between Ogres skeletal animating system and the input from Kinect sensor. When an instance of the object is initialized, it is given a mesh file. After the mesh is loaded with the skeleton data, the bone list is iterated through, given the initial orientation uniquely for each animatable bone with the Kinect sensor.

The animatable bones are set to be manually controlled, not inherit orientation from parents, given the initial orientations. At that state, the bone is set to initial state, to be reset at every frame with updated orientations. The non-animated bones are left to be automatically controlled and inherit orientation in

order to keep them aligned with their parent bones.

5.2 Animation

Every frame, the orientation information from the bones are extracted in 3x3 matrix form., along with the confidence of the sensor in that orientation. If the confidence is less than 0.5 in 1, the bones are left as they were in the previous frame to avoid unnatural movements.

The matrix from the Kinect is row major, whereas Ogre works with column major matrices. The transpose matrix is given to Ogre, turned into a quaternion and converted to local coordinate space. The updated orientation is fed into the bone, after it is multiplied with the initial orientation. The root bone is translated in local space in the end, to simulate the translation, above rotation.

This technique is used with the top-part of the dress as well, and it will be used for jackets, shirts and other clothing ware which do not leave the body surface by a great distance.

5.3 Algorithm

I used linear weighted skin blending [4] in order to simulate deformation on the characters skin. The effects of four bones with different weights are combined linearly to change the positions and normal of vertices.

5.3.1 The bone transformation Pseudo Code

The pseudo code below is executed during the pre-render cycle. The bone orientations are set to be ready for animation, deformation and rendering. At every render cycle, update Skeleton function is called, which automatically fetches the

coordinates from the Kinect and sets up the skeleton for vertex blending.

Algorithm 1: Bone transformation algorithm

```
1  function transformBone(bone)
       Input: A bone and the corresponding orientation matrix from Kinect
       Output: The same bone with updated orientation
2      q_I = initial orientation of bone;
3      q_N = relative orientation;
4      q_K = 3 × 3 Orientation Matrix From Kinect;
5      if kinect_confidence > 0.5 then
6          q_Q = toQuaternion(q_K);
7          q_N = toLocalSpace(q_Q);
8          q = q_N × q_I;
9          bone.orientation = q.normalise();
10 if user is new then
11     p_torso.initialize();                /* Initialize torso position */
12     ;
13 foreach bone do
14     if bone_orientation is new then
15         transformBone(bone);
16         skeleton.needsUpdate();
```

5.3.2 Deformation Pseudo Code

Algorithm 2 is executed prior to rendering, to update the vertices of the skin. The vertex blending function is where the deformation actually occurs.

(i):In my models, every vertex is assigned to at most 4 different bones. In order to speed up the deformation process, the transformation matrices for the corresponding matrices are collapsed into one. Collapsing is simply weighted addition of the four weighting matrices for a vertex.

Algorithm 2: Mesh update algorithm called at every frame.

```
1  function prepareBlendMatrices(mesh)
2      foreach bone do
3          bone.applyScale();
4          bone.applyTransform();
5          bone.applyOrientation();
6      i = 0;
7      foreach bone do
8          m[i] = bone.getTransformationMatrix4 × 4;
9          i++;
10     mapIndex=mesh.getIndexMap() ; /* Index Map contains the bone
       pointers for every vertex */
11     ;
12     i = 0;
13     foreach indexSet in mapIndex do
14         m_b[i] = indexSet[i] + m ;   /* Blend matrices are pointers to
           the individual bone matrices */
15         ;
16         i++;
17     return m_b;
18 function vertexBlend(m_b)
19     pos=*mesh.positions;        /* Pointer to the position matrix */
20     ;
21     norm=*mesh.normals;          /* Pointer to the normal matrix */
22     ;
23     b_i =*mesh.blendIndices; /* Pointer to the blend index matrix */
24     ;
25     b_w =*mesh.blendWeights; /* Pointer to the blend weight matrix
       */
26     ;
27     foreach 4 vertices in pos do
28         foreach vertex in 4 vertices do
29             m[1, 2, 3, 4] = m_b[b_i[vertex]] ;          /* Weighting Bones */
30             ;
31             m_c[j] = collapseMatrix(m, b_w[vertex])(i);
32         pos[4vertex] = m_c × pos[4vertex]
           norm[4vertex] = m_c × norm[4vertex]
33 if skeleton needs update then
34     m_b = prepareBlendMatrices(skeleton.mesh);
35     vertexBlend(m_b);
```

5.3.2.1 Matrix Operations

After four sets of matrix for four vertices are collapsed into four matrices, the following matrix operations are performed. 4 vertices are processed together in order to fully utilize the matrix multiplication features. Let us take the weighted matrix for vertex i:

$$M_i = \begin{bmatrix} m^i_{00} & m^i_{01} & m^i_{02} & m^i_{03} \\ m^i_{10} & m^i_{11} & m^i_{12} & m^i_{13} \\ m^i_{20} & m^i_{21} & m^i_{22} & m^i_{23} \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{5.1}$$

Note that this matrix is a linear combination of four linear transformation matrices from four weighting bones, hence the 3^{rd} row is by default [0 0 0 1].

We also have the initial positions for the 4 vertices as $[p^i_x\ p^i_y\ p^i_z\ 1].3^{rd}$ value, which is one is not included in the vertex buffer and it will not be taken into account with the calculations.

In order to generate an efficient SIMD process, I will perform 4x3 dot product calculations in each instruction. Dot products are commanded in the machine language, which makes it more preferable to higher level matrix calculations. To acquire the simulated x coordinates for the four vertices, we combine the first rows of all collapsed matrices for all and transpose it:

$$M_T = \begin{bmatrix} m^0_{00} & m^1_{00} & m^2_{00} & m^3_{00} \\ m^0_{01} & m^1_{01} & m^2_{01} & m^3_{01} \\ m^0_{02} & m^1_{02} & m^2_{02} & m^3_{02} \\ m^0_{03} & m^1_{03} & m^2_{03} & m^3_{03} \end{bmatrix} \tag{5.2}$$

To coincide with the transposed matrix, I also create a position matrix with positions from all vertices:

$$P_T = \begin{bmatrix} p_x^0 & p_x^1 & p_x^2 & p_x^3 \\ p_y^0 & p_y^1 & p_y^2 & p_y^3 \\ p_z^0 & p_z^1 & p_z^2 & p_z^3 \end{bmatrix} \tag{5.3}$$

Next, the matrices M_T and P_T are multiplied in a row-by-row fashion and summed together:

$$d_x = M_{T0} \times P_{T0} + M_{T1} \times P_{T1} M_{T2} \times P_{T2} + M_{T3} \tag{5.4}$$

The result vector d_x is a 4x1 vector which has the post-blended vertex x-coordinates: $[P_x^0\ P_x^1\ P_x^2\ P_x^3]$. The same procedure is applied to the normal of a vertex. Process continues with the next set of four vertices.

5.4 Interaction Between the Body And Cloth

The movements of the user are passed on the pieces of cloth separately.

5.4.1 Non-Simulated Section

Non simulated parts of the clothes are the ones which do not get separated from the body most of the time. Most parts of our clothes usually stick to the body and experience the same deformation as our skin. In order to increase performance, detailed simulation on these parts are not run, instead they are deformed the same as the remained of human body. In the full-body dress I am working with, the part above the waist has a skeleton similar to the body and the information from the Kinect is passed on to this portion as well. It is treated as a part of the actual avatar.

5.4.2 Simulated Section

The movements of the body are transferred into the simulated section of the cloth in three main ways:

5.4.2.1 Transformation

The fixed vertices are transformed to match the remainder of the cloth. The transformation is done on the rendering level, the physx world experiences no difference in transformation manner. This process keeps the cloth aligned with the rest of the visible world.

5.4.2.2 Collision

The colliding body is updated and collided with the cloth. The colliding body consists of 16 spheres and the capsules connecting the spheres. The details of this body is explained in section 7.1. This process keeps the cloth out of the avatars way.

5.4.2.3 Inertia

The Inertia of transformations is passed onto the simulated cloth, increasing the realism. The passed on inertia comes from the rotation and the transformation of the root bone of the human skeleton. With this process, although there is no actual transformation in the physics world, the resulting inertia effects are visible on the cloth itself.

Chapter 6

Cloth Simulation

For the cloth simulation, I utilized the Nvidia PhysX engine, which provides cloth simulation framework along with collision detection.

6.1 Model Set Up

The mesh to be simulated is the bottom part of the modeled dress which consists of 3100 vertices. It is aligned precisely with the top part and the female body to be simulated realistically.

I needed to provide the PhysX framework with the vertex and topology information separately, instead of a binary file or an automatic parser. I implemented a Wavefront OBJ file parser myself to acquire the information exported from a modeling suite, and feed the data into the physics engine.

Other than the vertex and topology information, an inverse weight needs to be specified for every vertex. This information can be embedded into the wavefront object as a second set of texture coordinates. On the other hand, this method adds too much extra data to the file, which is not a major problem, although not necessary. Therefore I chose to embed this information into the model, by specifying the material names separately for vertex groups with different weights.

The vertices which have a non-zero inverse weight have the suffix Free attached to their material name. The vertices are selected manually and their materials are attached. The current simulation required only two set of weights, which can be seen in Figure 6.1. After the weight information is implemented, the model is exported as a Wavefront OBJ file along with the MTL, and parsed into the software.

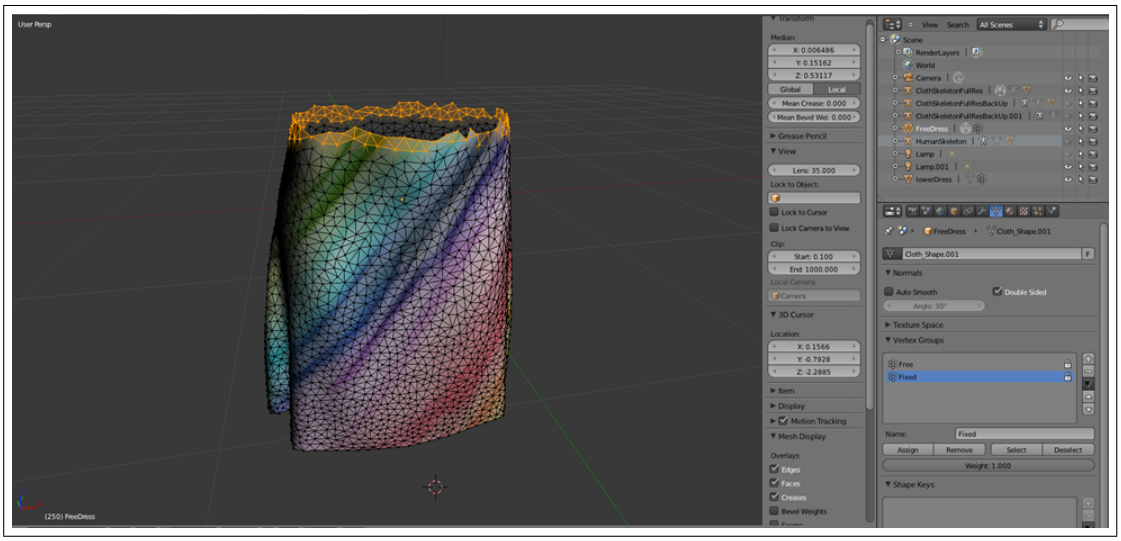

Figure 6.1: The fixed vertices of the cloth

6.2 The Initialization

After the model is parsed from the OBJ file, it is stored into C++ vectors initially, to be used in different frames. First, the OBJ file is converted into the Ogre native mesh format to be loaded and rendered. This simply includes feeding the vertex, topology and material information into the ogre as submeshes, to be combined as a mesh. Prior to creating the simulated cloth the PhysX engine needs to be initialized:

1. The foundation and the SDK objects are created.

2. In order to implement the GPU solver for the cloth and collision, the PhysX engine needs to bind with the CUDA context to deliver the GPU tasks. The

CUDA Context manager object is created and given to the SDK object.

3. Afterwards, the virtual floor and the environment are created with the gravity specified as 9.8 m/s2.

After the initialization, the cloth is loaded with specific stiffness, stretch, damping, friction, inertia, bending, and collision parameters. These parameters are decided after numerous experimentations in order to acquire the most realistic simulation results. After creating the environment and the virtual cloth, the collision spheres are created, which will be explained in chapter 7.

6.3 The Animation

Each frame, the workflow is as following:

1. The passed time since the last frame is added to the counter.
2. The Kinect Sensor is checked for new data. If the data is not new, next frame is called.
3. If there is an active user with the Kinect, the userID is set into the female body and the upper cloth meshes to be updated. If there are none, the bones are reset to their initial state.
4. The upper body mesh and the female body mesh are ordered to be updated. The update details can be found in chapter 5.2. This function returns the translation of the root node.
5. The returned vector from step 4 is used to translate the lower cloth handle. After the translation, the orientation is also updated with the root bone orientation.
6. The colliding human capsules are updated with the female body bone orientation. Also, the orientation and the position of the lower cloth handle

is synchronized with the virtual cloth. This input automatically introduces the inertia effect on the cloth.

7. In the end, the cloth is simulated as following:

   (a) PhysX is ordered to start the simulation on the GPU. Simulation algorithm details are explained in section 6.4.

   (b) The vertex data is read from the output, and the vectors of the OBJ object are updated, except the fixed mesh vertices. The reason for omitting the fixed vertices is to avoid unnatural bends and cracks on the fixed vertices.

   (c) The updated OBJ vectors are transformed into the Ogre Mesh buffers, to be rendered.

6.4 The Algorithm

The cloth simulation algorithm is based on the Position Based Dynamics, introduced by Mller et al. [11]. The main idea is to calculate the position of the particles from their previous positions and applying constraints on mutual distance and angle. The collision is also calculated as a force and applied to both particles. The approach is stable and efficient for real time applications.

The dynamics are stable as long as the constraint solvers converge. When this criteria is not met, anomalies show up, such as when a vertex is pulled too far away from the cloth.

The problem is parallelized by fibers, as SIMD process. With CUDA support, however, this is even parallelized more with SIMT processes, where each thread works on one fiber

6.4.1 Overall Structure

The position of a particle in the next time interval is acquired by performing Explicit Euler Integration over δt, where the velocity and the force are assumed to be constant during the interval. The pseudo code is given in Algorithm 3.

Algorithm 3: Position Based Dynamics

```
1  foreach vertices i do
2      initialize x_i = x_i^0, v_i = v_i^0, w_i = 1/m_i
3  while true do
4      foreach vertices i do
5          v_i ← v_i + Δt w_i f_ext(x_i)
6      dampVelocities(v_1, ..., v_N);
7      foreach vertices i do
8          p_i ← x_i + Δt v_i;
9      foreach vertices i do
10         generateCollisionConstraints(x_i → p_i)
11     while solverIterates do
12         projectConstraints(C_1, ..., C_{M+M_coll}, p_1, ..., p_N)
13     foreach vertices i do
14         v_i ← (p_i − x_i)/Δt;
15         x_i ← p_i;
16     velocityUpdate(v_1, ..., v_N);
```

The overall flow is to predict the next position and velocity (1-2), then perform the corrections by solving the constraints (3-12), update the position and velocities accordingly (13:15). Finally, the damping to the velocities is introduced in line 16.

The key issue in the simulation is the position correction due to the constraints. Each vertex is moved either towards or away from each other, the distance is scaled by the inverse mass of the vertices. If a vertex position needs to be fixed, the inverse weight should be set to zero

6.4.2 Constraints, Fibers and Sets

In order to simulate the cloth, all constraints should be solved, which is achieved through linearizing the non-linear problem. This linearization results in a sparse matrix problem. Although the problem is solvable in real-time, performance is increased further by pivoting vertical and horizontal constraints, solving separately. The vertical and horizontal constraints are divided in the input sense by fibers and sets. Fibers represent independent sets of connected constraints and sets are non-overlapping fibers, which are solved in parallel. In the implementation and mesh terms, a fiber is either a horizontal, vertical or a diagonal line, and the set is the collection of parallel lines. These fibers and sets are generated for both shear and stretch constraints by the PhysX mesh cooker, which auto-generates the fibers and sets from a given mesh topology (Figure 6.2).

Figure 6.2: Shear and Scretch Sets. The red and yellow lines are the fibers [6]

6.4.3 Set Solvers

There are two possible solvers to apply on fiber block(Set) which come with PhysX:

- The Gauss-Seidel solver continues along the fiber after completing a constraint solution and updating the results. This solver is easy to tweak and user, has low-cost for iteration, however the convergence factor is low due to sequential update, which results in a stretchy cloth.
- Semi-Implicit solver factorizes the tri-diagonal system with LDLT and solves

the overall system. This method preserves momentum since it is not sequential and converges ten times more than Gauss-Seidel solver. However, the matrices can be ill conditioned, which requires special treatment, iteration cost is higher and tweaking is more difficult.

The comparison of these solvers can be seen in Figure 6.3. In order to get the best performance from these solvers, they are applied on different Sets, taking advantage of them being solvable in parallel. Gauss-Seidel solver is used for horizontal stretch fibers and the shear fibers, which do not experience too much stretching, taking advantage of its low cost. Semi implicit solver is applied only to the vertical stretch fibers, which put up with most of the force (gravity) most of the time, resulting in a both fast and robust solution

Figure 6.3: Comparison of Gauss-Seidel and Semi-Implicit Solvers [6]

Chapter 7

Collision

Currently, the collision between the cloth and the body parts and the self-collision of cloth particles are implemented. This collision is handled by the PhysX engine.

7.1 Preparation and Settings

Cloth vertices can collide with pre-defined spheres, capsules or planes. In order to keep the program stable and running at real-time, it is important to find the balance between collision detail and performance.

The female body bone information is extracted from the input skeletal mesh class, and the colliding capsules are generated automatically, although the radii of the bones are specified manually. The collision information is specified in two arrays, one being the positions and radii of the spheres and the second defining pairs of spheres which form capsules. A total of 16 capsules and spheres are used in the whole collision model, which simulates all the movable bones of the female body (Figure 7.1). The collision data is prepared before the cloth creation, and given as a parameter.

Other than the defined collision spheres and capsules, the cloth naturally collides with the floor actor of the PhysX environment as well. This however,

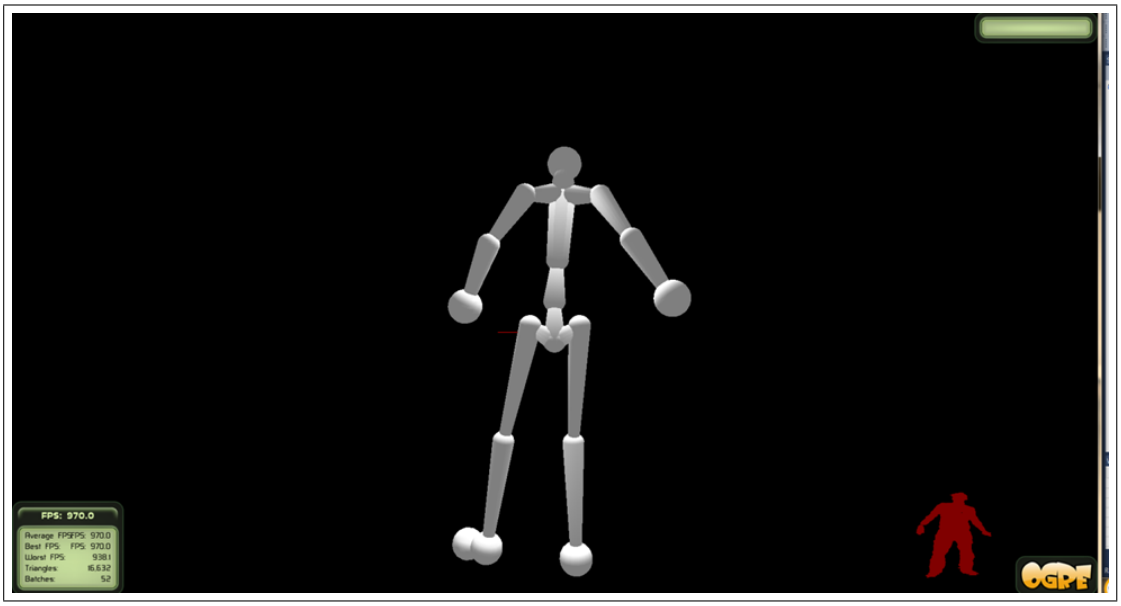

Figure 7.1: Character formed with collision spheres and capsules

introduces a problem: The models from Blender are exported as root joint coinciding with the origin. This is required for successful animation. However, in the PhysX environment, the floor is created automatically at (0,0,0), and the initial position of the lower dress is partly below floor, which produces unrealistic visuals. In order to overcome this issue, the lower cloth vertices are introduced into the software with increased y coordinates, in order to keep them above ground. In the rendering process, lower cloth is attached to a scene node with negative y offset equal to the boost in the vertices, which places the lower dress exactly where it should be. This process overcomes the floor collision problem.

Every frame, the collision sphere positions are updated with the data from the updated body skeleton, prior to the cloth simulation.

7.2 Algorithm

PhysX provides two options for collision detection: Discrete and Continuous Collision. I chose to work with the latter due to more robust results, although it takes twice as long to do the calculation.

The solution is performed for the trajectory of the capsule or sphere and the particle for frame interval. This approach is especially robust in fast motion, which is important since the motion is created in real time. The required solution is a 6th degree polynomial, which is approximated with quadratic equation. The pipeline is given in Figure 7.2.

The solution is performed on the GPU, which increases the parallel performance greatly, allowing for frame rates of 600+fps. The cloth is discretized as a triangle mesh, and since the collision is only detected on vertices, the density must be high enough in order to avoid penetration at the areas [6], [16].

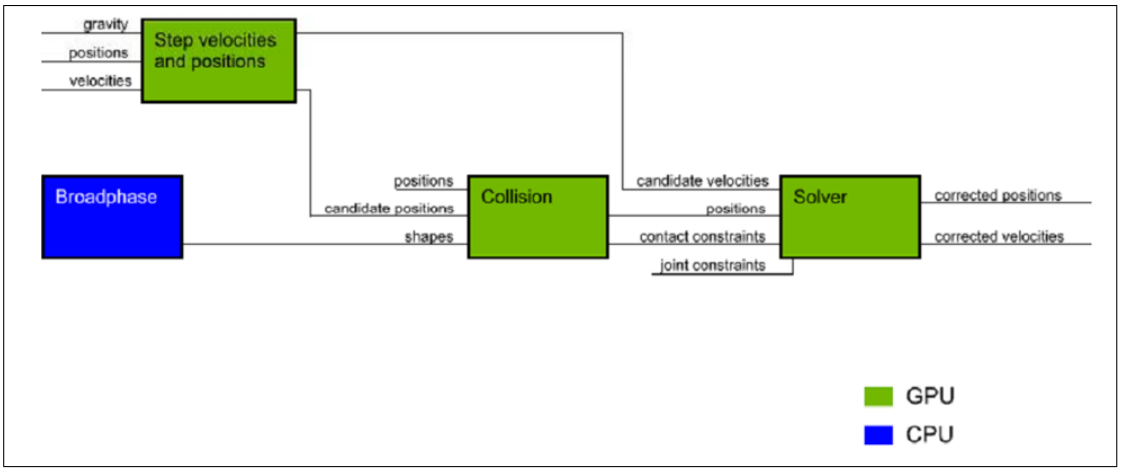

Figure 7.2: Collision Detection Pipeline [16]

Chapter 8

Cloth Resizing

My cloth resizing algorithm consists of three main steps:

1. Improve the raw depth map from Kinect by filtering.
2. Fit the contour on the user blob and perform measurements. Compare with known human body proportions to acquire required scaling parameters.
3. Perform the same measurements over a time frame in order to smooth the results. Scale the virtual avatar along with the cloth mesh prior to simulation.

8.1 Depth Map Optimization

At 30 fps, Kinect provides a depth map and a user map, both at 640x480 resolution. Depth map consists of distances with the sensor in millimeters.

The depth measurement of the Kinect is not very accurate compared to high end 3D depth systems like laser scanners. The accuracy of the depth value decreases quadratically. Error values for different distances is shown in Table 8.1.

For my application, the Kinect needs to be able to see the whole human body,

Distance of Point	1m	3m	5m
Error in measurement	0.5cm	1.5cm	4cm

Table 8.1: Kinect Depth Accuracy [5]

which requires at least 3m away from the sensor for a person with 1.7m height, resulting in an erroneous depth map. This problem can be seen in Figure 8.1.

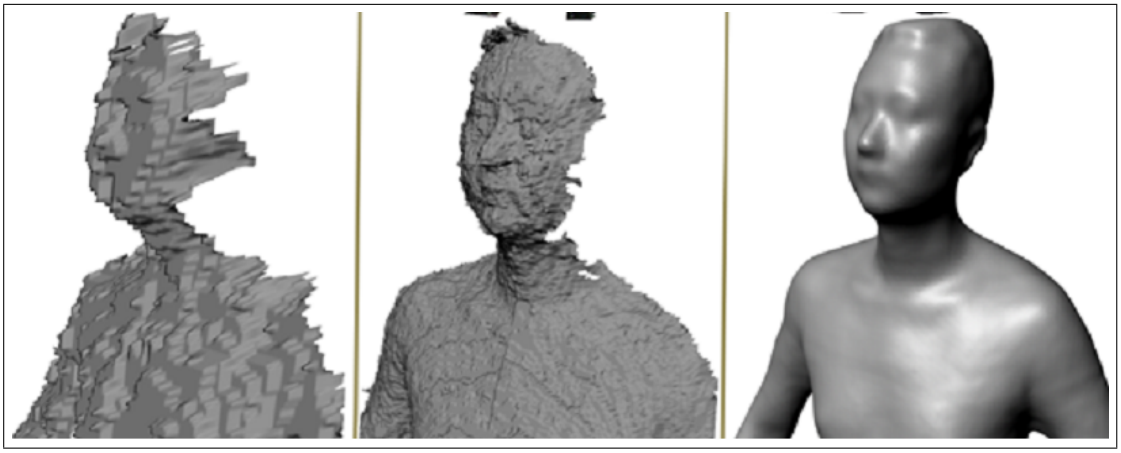

Figure 8.1: Depth Output From Kinect with Improved Results. The leftmost image is the raw output, the application of filters results in better performance in the other two images [15]. This improvement utilizes more than one depth streams.

In order to acquire a better depth map, I will perform the following operations:

Let us take the depth map D as a 640x480 matrix. Initially, the user pixels are extracted by a pixel-by-pixel comparison with the user map. User map is another acquired map from the sensor, with the same size as depth map. The value of a pixel is set to a non-zero value if the pixel belongs to a recognized user. In this case, we are only interested in one user, D_1 represents the depth pixels of the user we are interested in, whereas U_1 is the bit map of the user. Also, the non-user pixels must be filled with the mean value of the user pixels, in order to perform Gaussian filtering on the image.

$$D_1 = (D - (D \times U_1)) \times 1/n \times \sum_{i=0}^{n}((D \times U_1)_i + d \times U_1) \tag{8.1}$$

Next, we perform Gaussian filtering on the users depth map, to normalize and improve the quality of the depth map. The size and sigma parameters of the Gaussian filter will be varied in order to maximize the performance and the quality of the results.

$$D_G = D_1 * G \tag{8.2}$$

After these operations, we have a normalized and filtered depth map, which also has better planar values (x and y) since the holes due to depth stream will be filled.

Algorithm 4: Depth Map Optimization Algorithm

Input: Raw Depth Stream From Kinect
Output: Depth Stream With Patched Holes and Gaussian Optimization

```
1  depth_sum = 0 ;
2  n_user = 0;
3  for i from 0 to d_width do
4      for j from 0 to d_height do
5          if U(i, j) then
6              depth_sum = depth_sum + D(i, j);
7              n_user+ = 1;
8  depth_average = depth_sum/n_user ;
9  for i from 0 to d_width do
10     for j from 0 to d_height do
11         if not U(i, j) then
12             D(i, j) = depth_average;
13 for i from 0 to d_width do
14     for j from 0 to d_height do
15         if U(i, j) then
16             D(i, j) = D(i − m : i + m, j − n : j + m) * Gaussian(m, n, e);
17 return D
```

8.2 Parameter Measurement

In paremeter measurement, I will handle two objectives: To determine the optimal sizes of collision spheres for cloth simulation and the required scaling parameters for the cloth to optimally fit the user. It is important that these algorithms do not take more than a thirtieth seconds on a high-end consumer computer in order to keep the real time experience smooth, since there is an averaging over time is involved.

First step will be fitting spheres in various locations in the optimized body map. These fitted spheres will provide the radii for the collision spheres which will be used to simulate the cloth. Locations of the machine-provided user joints are shown in Figure 8.2. These are where spheres will be located.

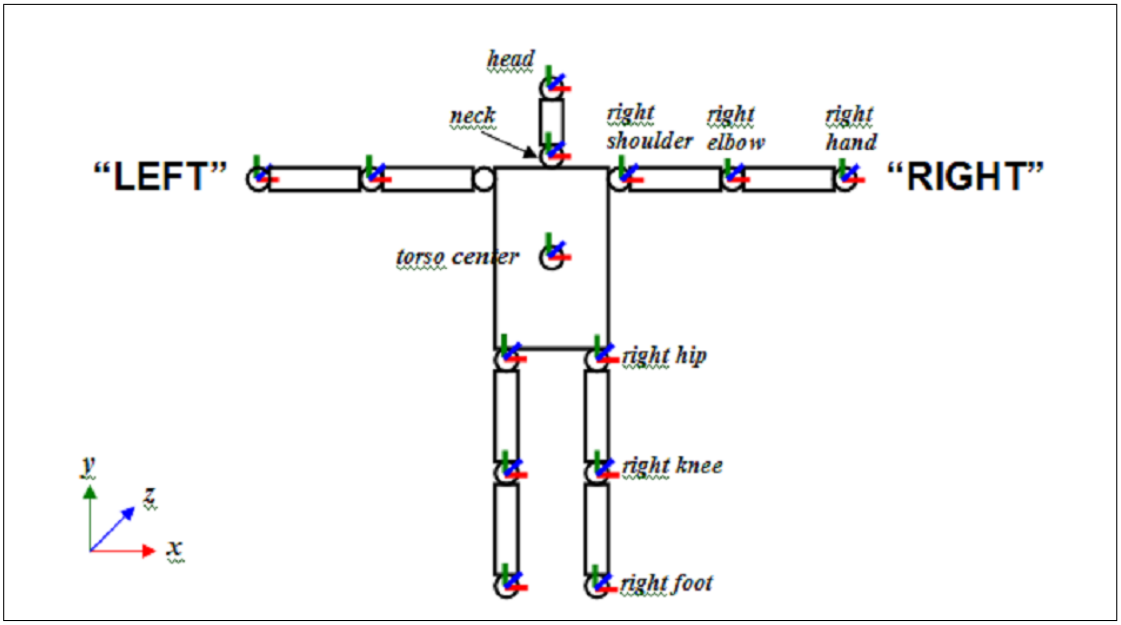

Figure 8.2: Human Joints provided by NITE

Sphere fitting algorithm will go as following:

1. Take vector J_i which represents the coordinates of the i^{th} joint. First, initialize the radius of the sphere by the difference of z-dimension with the

overlaying point in the depth map.

$$r_i^z = J_i^z - D^z(J_i^x, J_i^y) \tag{8.3}$$

2. Repeat the same process for the x and y dimensions in both negative and positive directions. Take the bigger radius. If there are no points on either side, set it to zero.

$$r_i^{x,y} = max(\| \pm J_i^{x,y} \mp D^{x,y}(J_i^{y,x}, J_i^z)\|) \tag{8.4}$$

3. 3. The radius of the sphere is equal to the minimum of these three values.

$$r_i = min(r_i^{x,y,z}) \tag{8.5}$$

8.3 Human Body Parameters

Next step will be acquiring the optimal scaling parameters for the cloth. The human body proportions to be used are shown in Table 8.2. In these proportions, the unit width and height are taken as the width and height of the head. The measurements source indicates how the measurement on the user will be performed: Joint Location means algorithm will make use of the joint locations provided by NITE, whereas depth map means the filtered depth map will be used. They can be used together in order to improve the performance. Some of these proportions are not standard enough to be used as references and vary greatly, such as hip width, and will be used directly from measurement.

Along with the ratios, the actual size in meters in height and width shall be measured and recorded as well, since the cloth needs to be scaled according to the user. In my initial approach, I will scale the whole cloth as a whole, with different parameters for three dimensions. If this approach proves unrealistic, I will process with different scaling parameters for different portions of the cloth, although this would not prove useful in a real fitting room, since most shops do not offer extensive customization.

Algorithm 5: Sphere Fitting Algorithm

Input: Optimized Depth Stream From Kinect
Output: Collision Sphere radii for each joint

foreach *joint* **do**
  $p = pos_{J_m}$;
  $r_z = \sqrt{P_z^2 - D_z(P_x, P_y)^2}$ **for** i **from** P_x **to** 0 **do**
    if $D(i, P_y)$ **equals** P_z **then**
      $r_x^- = i$;
      break;
  for i **from** P_x **to** $depth_width$ **do**
    if $D(i, P_y)$ **equals** P_z **then**
      $r_x^+ = i$;
      break;
  for j **from** P_y **to** 0 **do**
    if $D(P_x, j)$ **equals** P_z **then**
      $r_y^- = j$;
      break;
  for j **from** P_y **to** $depth_height$ **do**
    if $D(P_x, j)$ **equals** P_z **then**
      $r_y^+ = j$;
      break;
  $r_m = min(r_z, r_x^-, r_x^+, r_y^-, r_y^+)$
return $(r_0, r_1 \dots r_n)$

Distance	**Width**	**Height**	**Measure Source**
Head	1w (1)	1h (2)	Depth Map+Joint Location
Body Height	-	7 (3)	Depth Map
Hip Height	-	4 (4)	Joint Location
Elbow-Fingertip	-	2 (5)	Depth Map+Joint Location
Wrist to Fingertip	-	1 (6)	Depth Map+Joint Location
Shoulder Width	3 (7)	-	Depth Map+Joint Location
Hip Width	- (8)	-	Depth Map
Torso Height	-	- (9)	Joint Location

Table 8.2: Human Body Proportions [1]. Numbers in parenthesis represent the lines on Figure 8.3.

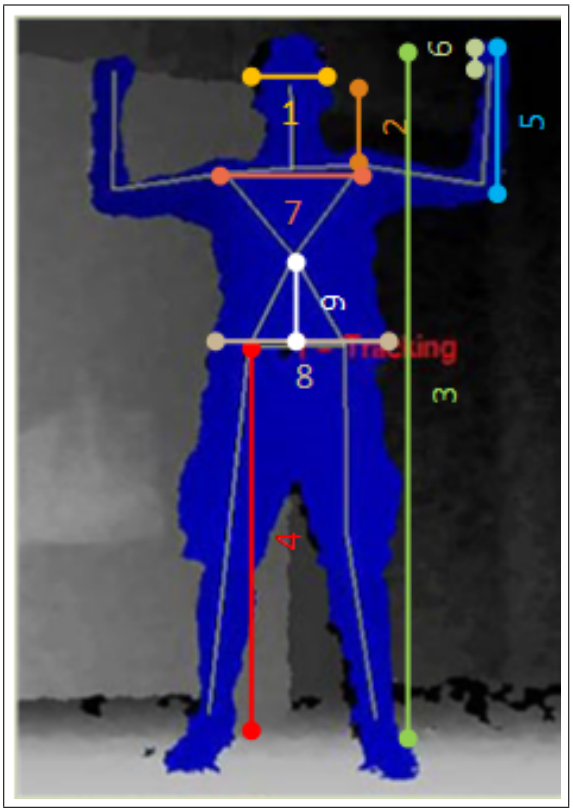

Figure 8.3: Proportions on the Body

As the types of cloth focus on different portions of the body, the human body proportions from different areas should not affect the scaling parameters equally. Therefore, the main body parameters for different types of clothes are listed in Table 3.

The parameter estimation algorithm will go as following:

1. For a particular cloth, take the primary measured proportion as P_i^0. This will be the measured dimension of said proportion. This process will be

Type Of Cloth	**Primary Height Proportions**	**Primary Width Proportions**
Trousers	Hip Height	Hip Width
Long Sleeves	Body Height	Elbow-Fingertip Height,Shoulder W
Short Sleeves-Sleeveless	Torso Height	Shoulder Width

Table 8.3: Primary Proportions for Different Cloth Types

repeated for width (W) and height (H).

2. With all the other measured proportions, calculate the estimated value of P_i as P_i^j. Here, R denotes the ratio from Table 8.2.

$$W, H_i^j = W, H_j \times R_i^j \tag{8.6}$$

3. Find the optimized main parameter width as the average:

$$W, H_i = 1/(n+1) \times \sum_{j=0}^{n} W, H_i^j \tag{8.7}$$

After finding the optimized main parameter in meters, it can be used to scale the virtual cloth by calculating the ratio.

Algorithm 6: Cloth Resizing Algorithm

$t_{proportion} = import(Table8.2)$;
$t_{primary} = import(Table8.3)$;
$ct = cloth_{type}$;
$width_{main} = t_{proportion}.width(ct)$;
$width_{sum} = 0$;
$count_{effector} = 0$;
foreach *width* **in** $t_{proportion}$ **do**
    $w_i = measure(p_i)$;
    $w_i^j = w \times t_{proportion}.ratio(p_i, parameter_{main})$;
    $width_{sum} = width_{sum} + w_i^j$;
    $count_effector++$;
$width_{weighted} = width_{sum}/count_effector$ $x_s = width_{weighted}/width_{cloth}$;
$height_{main} = t_{proportion}.height(ct)$;
$height_{sum} = 0$;
$count_{effector} = 0$;
foreach *height* **in** $t_{proportion}$ **do**
    $h_i = measure(p_i)$;
    $h_i^j = h \times t_{proportion}.ratio(p_i, parameter_{main})$;
    $height_{sum} = height_{sum} + h_i^j$;
    $count_effector++$;
$height_{weighted} = height_{sum}/count_effector$ $y_s = height_{weighted}/height_{cloth}$;
return (x_s, y_s)

Bibliography

[1] W. B. Body proportions in art. http://www.worsleyschool.net/socialarts/body/proportions.ht 2012.

[2] G. Bradski. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*, 2000.

[3] K.-M. G. Cheung, S. Baker, and T. Kanade. Shape-from-silhouette across time part ii: Applications to human modeling and markerless motion tracking. *Int. J. Comput. Vision*, 63(3):225–245, July 2005.

[4] L. Kavan and J. Zara. Real-time skin deformation with bones blending. In *WSCG Short Papers Proceedings*, 2003.

[5] K. Khoshelham and S. O. Elberink. Accuracy and resolution of kinect depth data for indoor mapping applications. *Sensors*, 12(2):1437–1454, 2012.

[6] T.-Y. Kim. Character clothing in physx-3, 2011.

[7] T. Knot. Ogre. open source 3d rendering engine. http://www.ogre3d.org/, 2012.

[8] LadyJewell. Antonia sundress. sharecg. http://www.sharecg.com/v/54636/gallery/5/3D-Model/Antonia-Sundress, 2012.

[9] Mmava. Free female base mesh. zbrush central. http://www.zbrushcentral.com/showthread.php?49053-Free-female-base-mesh-(nudity)., 2012.

[10] J. Moan. Documentation architecture. ogre- object oriented rendering engine. http://www.ogre3d.org/tikiwiki/tiki-index.php?page=Documentation+Architecture/, 2012.

[11] M. Muller, B. Heidelberger, M. Hennix, and J. Ratcliff. Position based dynamics. *J. Vis. Comun. Image Represent.*, 18(2):109–118, Apr. 2007.

[12] OpenNI. Programmer guide-openni. http://openni.org/Documentation/ProgrammerGuide.ht 2012.

[13] PrimeSense. Nite. primesense natural interaction. http://www.primesense.com/en/nite, 2012.

[14] M. Tang. Recognizing hand gestures with microsofts kinect. Technical Report MSU-CSE-00-2, Department of Computer Science, Stanford University, January 2011.

[15] J. Tong, J. Zhou, L. Liu, Z. Pan, and H. Yan. Scanning 3d full human bodies using kinects. *IEEE Transactions on Visualization and Computer Graphics*, 18(4):643–650, Apr. 2012.

[16] R. Tonge-Nvidia. Collision detection in physx, 2010.

Appendix A

Rhinoplasty Application

We conducted two different experiments to apply our solution in the area of rhinoplasty. In the experiments, we correct the form of misshapen noses (see Figure A.1). We compare the accuracy of the deformations for linear and nonlinear FEMs. The results for these experiments are interpreted by comparing displacement amounts for the force applied nodes and all the nodes.

A.1 Experiment 1

The first experiment is conducted with a head mesh that has 6709 nodes and 25722 tetrahedral elements (see Figure A.2 (a)). Number of tetrahedral elements are very high. However, all the operations are done in the nose area with 1458 tetrahedral elements. To simplify the calculations, stationary tetrahedral elements are not taken into account. Figure A.2 (b) shows that the head mesh is constrained from the blue nodes, and is pushed upwards at the green nodes. This experiment is conducted to observe the different effects of the nonlinear and the linear FEM deformations over the nose. Linear FEM produced high amount of displacement at the upper part of the nose (see Figure A.2 (c)). As a result of that, the node displacement error becomes 64.92%. It can be observed that the

Figure A.1: The perfect nose.

nose was deformed more realistically and smoothly with nonlinear FEM (see Figure A.2 (d)). The nose is more collapsed inwards with linear FEM, whereas its structure is better preserved with nonlinear FEM and the overall shape is more similar to a perfect nose than linear FEM. Although, nearly 6000 nodes are constrained, the overall nodal displacement error is 3.88%.

(a)

(b)

(c)

(d)

Figure A.2: Experiment 1: (a) Initial misshapen nose. (b) Head mesh is constrained from the blue nodes, and is pushed upwards at the green nodes. (c) Linear FEM Solution: left: wireframe surface mesh with nodes, right: shaded mesh with texture. (d) Nonlinear FEM Solution: left: wireframe surface mesh with nodes, right: shaded mesh with texture.

A.2 Experiment 2

The second experiment is conducted with mesh similar to the one used in the first experiment. However, it has 7071 nodes and 27020 tetrahedral elements due to different shape of the nose and tetrahedralization (see Figure A.3 (a)). To simplify the calculations, stationary tetrahedral elements are not taken into account. 4511 tetrahedra are included in the calculations. A.3 (b) shows that the head mesh is constrained from the blue nodes, and is pushed upwards at the green nodes. This experiment is conducted to observe the different effects of the nonlinear and the linear FEM deformations over the nose. Linear FEM produced high amount of displacement at the lower part of the nose (see A.3 (c)). Moreover, the nose is nearly collapsed inwards that is far away from the perfect nose. As a result of that, the node displacement error becomes 96.03%. With nonlinear FEM, the nose is deformed more realistically and smoothly (see A.3 (d)). The nose is more collapsed inwards with linear FEM, whereas its structure is better preserved with nonlinear FEM and the overall shape is more similar to a perfect nose than linear FEM. Although, nearly 5000 nodes are constrained, the overall nodal displacement error is 11.19%.

(a) (b)

(c) (d)

Figure A.3: Experiment 2: (a) Initial misshapen nose. (b) Head mesh is constrained from the blue nodes, and is pushed upwards at the green nodes. (c) Linear FEM Solution: left: wireframe surface mesh with nodes, right: shaded mesh with texture. (d) Nonlinear FEM Solution: left: wireframe surface mesh with nodes, right: shaded mesh with texture.