Università degli Studi di Firenze

Computer Graphics and 3D

Kinect skeleton extraction and animation in a simple 3D world

Federico Màlato

Academic Year 2018-19

Outline

Introduction

3D scanners are one of the most used ways of gathering 3D data, which are becoming more and more popular for a wide spread of task, like:

Introduction

3D scanners are one of the most used ways of gathering 3D data, which are becoming more and more popular for a wide spread of task, like:

- Medicine

Introduction

3D scanners are one of the most used ways of gathering 3D data, which are becoming more and more popular for a wide spread of task, like:

- Medicine
- Chemistry and biology

Introduction

3D scanners are one of the most used ways of gathering 3D data, which are becoming more and more popular for a wide spread of task, like:

- Medicine
- Chemistry and biology
- Industrial process

Introduction

3D scanners are one of the most used ways of gathering 3D data, which are becoming more and more popular for a wide spread of task, like:

- Medicine
- Chemistry and biology
- Industrial process
- Machine learning

Introduction

3D scanners are one of the most used ways of gathering 3D data, which are becoming more and more popular for a wide spread of task, like:

- Medicine
- Chemistry and biology
- Industrial process
- Machine learning
- And many others

Introduction

3D scanners are one of the most used ways of gathering 3D data, which are becoming more and more popular for a wide spread of task, like:

- Medicine
- Chemistry and biology
- Industrial process
- Machine learning
- And many others

So, it's useful to know how to get those data and how to handle them.

To achieve my goal, I've been using:

- A Microsoft Kinect v2
- C++
- MatLab
- OpenGL 4.6 with some other libraries from the Khronos suite
- GitHub (code available at [2])
- A body
- Patience

Goal

The goal of my project was to implement a working C++ program that, given the **position** of some key joints of a human body, could replicate that pose in a simple 3D envionment generated through OpenGL.

Goal

The goal of my project was to implement a working C++ program that, given the **position** of some key joints of a human body, could replicate that pose in a simple 3D envionment generated through OpenGL.

Since OpenGL is an old library, I've been using some third party extensions in order to simplify a bit the tedious work like defining polygons and regular shapes. In particular, I've been using **GLUT** and **freeGLUT** to use already defined shapes like spheres, **GLM** to help with the math and **glad** mostly for the technical part like handling the moving camera and so on.

Project structure

Due to some conflicts between Kinect and C++, I've been forced to split the program in two parts. Those two parts are connected by a **.csv** file that is **written** from the MatLab script and **read** from the C++ program. Hence, the project is structured between two main programs:

- The first program uses **C++** and shows the 3D world and the skeleton moving in it
- The second program is written in **MatLab** and allows to acquire the 3D skeleton data in both batch and real time mode.

MatLab

- For the MatLab script, I've been using an **already existing** script available at [1].

MatLab

- For the MatLab script, I've been using an **already existing** script available at [1].
- Starting from this, I have edited the *bodyDemo.m* file in order to store the joints of the current frame in a **.csv** file.

MatLab

- For the MatLab script, I've been using an **already existing** script available at [1].
- Starting from this, I have edited the *bodyDemo.m* file in order to store the joints of the current frame in a **.csv** file.
- The script stores in a **slightly different** way the data for the non realtime skeleton and the realtime one: while in the first case the new position is appended at the end of the file, the other one uses this file as a temporary buffer. Hence, the new position overwrites the previous one.

```
buffer(1).Position = bodies(1).Position;
buffer(1).Orientation = bodies(1).Position;
buffer(1).TrackingState = bodies(1).TrackingState;

cellBuffer = struct2cell(buffer);
tableBuffer = cell2table(cellBuffer(1:end,:));
writetable(tableBuffer, 'C:/Users/fredd/CLionProjects/3D_avatar/KinectJointsRealtime.csv')
```

Figure: As for the code, the difference between the realtime and the non realtime version of the script is minimal: the non realtime script saves in a *KinectJoints.csv* file and append each new position after the previous one, using an *i* index instead of the 1 constant.

MatLab

Also, the *bodyDemo.m* opened two windows showing the depth image and the color image with the skeleton. Since those windows **slowed** the computation (the Kinect recorded at about 5 FPS), I've disabled the windows and created another script, *bodyDemoWithWindow.m* with the exact same content but with the windows enabled. Both of the scripts have been tested and are fully working.

C++

The core of the project is surely represented by the C++ program.

C++

The core of the project is surely represented by the C++ program. The code is organized around a *main.cpp* file, four classes, a *utils.h* file and a *Shader* folder, that contains a very basic implementation of a vertex and a fragment shader written in GLSL.

C++

The core of the project is surely represented by the C++ program. The code is organized around a *main.cpp* file, four classes, a *utils.h* file and a *Shader* folder, that contains a very basic implementation of a vertex and a fragment shader written in GLSL.
Since each file is pretty dense, we will take a rapid look at each of it in the next few slides.

C++ - main.cpp

The file starts with a bunch of function declaration. The matching definitions are stored in the *utils.h* file, in order to put a limit on the length of the file.

C++ - main.cpp

The file starts with a bunch of function declaration. The matching definitions are stored in the *utils.h* file, in order to put a limit on the length of the file.
Before the **main()** function, there are also a bunch of global variables. I've come up with using them because I needed most of them in various parts of the code and, since there are no race conditions, it was the best way I could find to spare some memory.

C++ - main.cpp

The file starts with a bunch of function declaration. The matching definitions are stored in the *utils.h* file, in order to put a limit on the length of the file.
Before the **main()** function, there are also a bunch of global variables. I've come up with using them because I needed most of them in various parts of the code and, since there are no race conditions, it was the best way I could find to spare some memory.
In the **main()** function, we set the openGL environment and initialize the vertex and the fragment shaders as C++ Objects. Also, if the running mode is set as "not realtime", an interpolation on the positions stored in the *KinectJoints.csv* file is performed immediately.

C++ - main.cpp

The file starts with a bunch of function declaration. The matching definitions are stored in the *utils.h* file, in order to put a limit on the length of the file.
Before the **main()** function, there are also a bunch of global variables. I've come up with using them because I needed most of them in various parts of the code and, since there are no race conditions, it was the best way I could find to spare some memory.
In the **main()** function, we set the openGL environment and initialize the vertex and the fragment shaders as C++ Objects. Also, if the running mode is set as "not realtime", an interpolation on the positions stored in the *KinectJoints.csv* file is performed immediately.
While I've defined a custom *draw()* function for each object, I've decided not to do that for the world grid and the coordinate system. I just thought that they should have behaved like a singleton and so I've made it hard to create another instance of them.

C++ - main.cpp

Finally, the main loop starts. Here, the various *draw()* functions are called. While drawing the skeleton is pretty simple, drawing the body of it requires multiple calls to the same functions with different parameters. That's why it looks a little messy.

```cpp
// ... and the same goes for the cylinders
for(int j = 0; j < 48; j += 2) {
    distance = (float)sqrt(pow(joints[skeletonIndices[j + 1]]->getX() - joints[skeletonIndices[j]]->getX(), 2) +
            pow(joints[skeletonIndices[j + 1]]->getY() - joints[skeletonIndices[j]]->getY(), 2) +
            pow(joints[skeletonIndices[j + 1]]->getZ() - joints[skeletonIndices[j]]->getZ(), 2));
    if(skeletonIndices[j + 1] == 6 || skeletonIndices[j + 1] == 10) {
        bRadius = 0.05f;
        tRadius = 0.1f;
    }
    else if(skeletonIndices[j + 1] == 11 || skeletonIndices[j + 1] == 24 || skeletonIndices[j + 1] == 23 ||
            skeletonIndices[j + 1] == 7 || skeletonIndices[j + 1] == 21 || skeletonIndices[j + 1] == 22) {
        bRadius = 0.05f;
        tRadius = 0.05f;
    }
    else if(skeletonIndices[j + 1] == 17 || skeletonIndices[j + 1] == 13) {
        bRadius = 0.115f;
        tRadius = 0.15f;
    }
    else if(skeletonIndices[j + 1] == 14 || skeletonIndices[j + 1] == 18) {
        bRadius = 0.09f;
        tRadius = 0.115f;
    }
    else{
        bRadius = 0.1f;
        tRadius = 0.1f;
    }
    if(skeletonIndices[j + 1] == 4 || skeletonIndices[j + 1] == 8 || skeletonIndices[j + 1] == 12 ||
            skeletonIndices[j + 1] == 16 || skeletonIndices[j + 1] == 1 || skeletonIndices[j + 1] == 0) {
        color = {0.0f, 1.0f, 0.0f};
    }
```

Figure: It looks way harder than it really is...

```cpp
// Since we need a bunch of different spheres (depending on the joint we're using), we need to use this bad
// if-else statment...
for(int i = 0; i < joints.size(); i++) {
    if(i != 1 && i != 2 && i != 3 && i != 22 && i != 24 && i != 11 && i != 7 && i != 21 && i != 23 && i != 6 && i != 10) {
        drawSphere({0.1, 0.1, 0.7},
                   {joints[i]->getX() + 6, joints[i]->getY() + (float) 2.5, joints[i]->getZ() + 2}, 0.15);
    }
    else if(i == 3) {
        drawSphere({0.1, 0.1, 0.7},
                   {joints[i]->getX() + 6, joints[i]->getY() + (float) 2.5, joints[i]->getZ() + 2}, 0.3);
    }
    else if(i == 22 || i == 24 || i == 11 || i == 7 || i == 21 || i == 23 || i == 6 || i == 10) {
        drawSphere({0.1, 0.1, 0.7},
                   {joints[i]->getX() + 6, joints[i]->getY() + (float) 2.5, joints[i]->getZ() + 2}, 0.075);
    }
    else if(i == 0 || i == 1) {
        drawSphere({0.0, 1.0, 0.0},
                   {joints[i]->getX() + 6, joints[i]->getY() + (float) 2.5, joints[i]->getZ() + 2}, 0.1);
    }
    else {
        drawSphere({1.0, 1.0, 0.0},
                   {joints[i]->getX() + 6, joints[i]->getY() + (float) 2.5, joints[i]->getZ() + 2}, 0.1);
    }
}

// ... and the same goes for the cylinders
for(int j = 0; j < 48; j += 2) {
    distance = (float)sqrt(pow(joints[skeletonIndices[j + 1]]->getX() - joints[skeletonIndices[j]]->getX(), 2) +
            pow(joints[skeletonIndices[j + 1]]->getY() - joints[skeletonIndices[j]]->getY(), 2) +
            pow(joints[skeletonIndices[j + 1]]->getZ() - joints[skeletonIndices[j]]->getZ(), 2));
```

Figure: ... I swear it.

C++ - Shader

Shader is one of the four classes of the project. I've decided to create a Shader object since, while understanding them through the course slides and various tutorials, it seemed to me the best way to access and use them.

C++ - Shader

Shader is one of the four classes of the project. I've decided to create a Shader object since, while understanding them through the course slides and various tutorials, it seemed to me the best way to access and use them. The Shader class contains a Shader() constructor method that initializes a vertex, a fragment and (optionally) a geometry shader, given the file paths of each one of them. To create them, it uses the standard OpenGL pipeline directives, like *glCompileShader(), glAttachShader()* and so on, without messing the **main()** code.

C++ - Shader

Shader is one of the four classes of the project. I've decided to create a Shader object since, while understanding them through the course slides and various tutorials, it seemed to me the best way to access and use them. The Shader class contains a Shader() constructor method that initializes a vertex, a fragment and (optionally) a geometry shader, given the file paths of each one of them. To create them, it uses the standard OpenGL pipeline directives, like *glCompileShader(), glAttachShader()* and so on, without messing the **main()** code.
Also, the class provides some basic methods to set an *Uniform* variable (be it a float, a vector or a matrix) and a *checkCompileErrors()* method that prints the content of the GLShader's infoLog if something goes wrong during the compilation.

C++ - Shaders code

```
#version 460 core

layout (location = 0) in vec3 aPos;
layout (location = 1) in vec3 aColor;

out vec3 actualColor;

uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;

void main() {
    gl_Position = projection * view * model * vec4(aPos, 1.0);
    actualColor = aColor;
}
```

Figure: Vertex shader code.

```
#version 460 core

in vec3 actualColor;
in vec2 TexCoord;

out vec4 FragColor;

void main() {

    FragColor = vec4(actualColor, 1.0);

}
```

Figure: Fragment shader code.

- Since the polygons I had to draw were really basic, I needed a really basic couple of shaders

C++ - Shaders code

```
#version 460 core

layout (location = 0) in vec3 aPos;
layout (location = 1) in vec3 aColor;

out vec3 actualColor;

uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;

void main() {
    gl_Position = projection * view * model * vec4(aPos, 1.0);
    actualColor = aColor;
}
```

Figure: Vertex shader code.

```
#version 460 core

in vec3 actualColor;
in vec2 TexCoord;

out vec4 FragColor;

void main() {

    FragColor = vec4(actualColor, 1.0);

}
```

Figure: Fragment shader code.

- Since the polygons I had to draw were really basic, I needed a really basic couple of shaders
- Others polygon, drawn with the GLU functions, also didn't required a very powerful shader

C++ - Camera

- I've decided to introduce a moving camera mostly for a practical reason: when it came to look at a drawn object, to see if there was any error, I could only try to rotate it or setting the Point of View (POV) manually.

C++ - Camera

- I've decided to introduce a moving camera mostly for a practical reason: when it came to look at a drawn object, to see if there was any error, I could only try to rotate it or setting the Point of View (POV) manually.
- Since it was incredibly awful and boring, the moving camera seemed a good solution.

C++ - Camera

- I've decided to introduce a moving camera mostly for a practical reason: when it came to look at a drawn object, to see if there was any error, I could only try to rotate it or setting the Point of View (POV) manually.
- Since it was incredibly awful and boring, the moving camera seemed a good solution.
- The *Camera* object can be controlled using the W, A, S, D keys and its field of view can be rotated just moving the mouse
- Also, it can't go below the grid

C++ - Position and Joint

- Those two classes have helped me a lot while drawing the skeleton, and that's mostly why I've tried to generate them

C++ - Position and Joint

- Those two classes have helped me a lot while drawing the skeleton, and that's mostly why I've tried to generate them
- At the start, when the program read the *KinectJoints.csv* file, is stored each joint on a *std::vector of floats*.

C++ - Position and Joint

- Those two classes have helped me a lot while drawing the skeleton, and that's mostly why I've tried to generate them
- At the start, when the program read the *KinectJoints.csv* file, is stored each joint on a *std::vector of floats*.
- Then, since I had to compute a bunch of joints at the same time, I had to deal with a *std::vector of std::vectors of floats*...
- Even using *glm::vec3*, when it came up to draw the lines between the joints I couldn't understand my own code.

C++ - Position and Joint

- Those two classes have helped me a lot while drawing the skeleton, and that's mostly why I've tried to generate them
- At the start, when the program read the *KinectJoints.csv* file, is stored each joint on a *std::vector of floats*.
- Then, since I had to compute a bunch of joints at the same time, I had to deal with a *std::vector of std::vectors of floats*...
- Even using *glm::vec3*, when it came up to draw the lines between the joints I couldn't understand my own code.
- So, I created the Joint class with three attributes **x, y,** and **z** with basic methods.
- Also, the *Position* class has been defined as a vector of 25 joints, also with basic methods.
- Even if those class seemed pretty useless at the start, the latest usage as pointers have proved them useful.

Result and issues

- I've managed to make the program work with both the realtime and the non realtime approach.
- Unfortunately, due to some setup issues (mostly Linux) and the fact that I don't have a NVidia GPU on the laptop, I'm not able to show the program on the fly.
- So, I've recorded two small clips to show what my project can achieve.

Result and issues

- I've managed to make the program work with both the realtime and the non realtime approach.
- Unfortunately, due to some setup issues (mostly Linux) and the fact that I don't have a NVidia GPU on the laptop, I'm not able to show the program on the fly.
- So, I've recorded two small clips to show what my project can achieve.
- But first, I'd like to discuss a bit further the results in both cases.

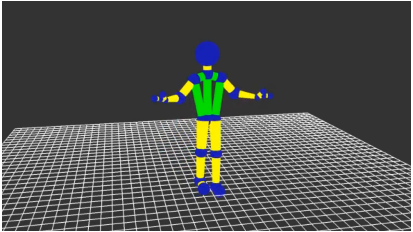

Figure: Non realtime body.

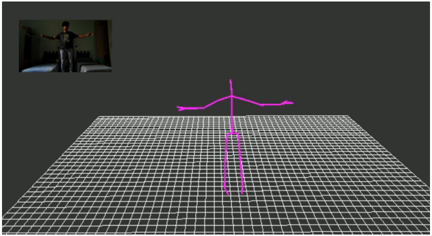

Figure: Realtime skeleton.

Results and issues - non realtime

- I've recorded a short videoclip using the MatLab script and then read the output file with the C++ program.
- From MatLab I got about **120 frames for a 25-seconds video.**
- To run things a bit more smoothly, I used a linear interpolation between my keyframes.
- After that, I got about **1000 frames.**
- Still, there are some weird movements of some joints. After investigating this, I've discovered that it was because of a bad estimation of the pose by the Kinect.
- While working with the non realtime part, things ran pretty smooth.
- After I had figured out how to link MatLab and C++, the skeleton showed up almost immediately and you can see the final result in the following clip

Results and issues - realtime

- The realtime part was harder
- First of all, the MatLab and the C++ performed some asynchronous read/write operations over the *KinectJointsRealtime.csv* file.
- This lead to a segmentation error: at some point, C++ would read an incomplete pose from the file
- I've tried many and many solutions, and a good one has been proved to store the last complete position reading in a global variable **lastKnownPos**: if there's an error while reading, *lastKnownPos* is drawn instead of the current one.
- Still, when drawing the body, I noticed empirically that the asynchrony was particularly bad.
- So, I decided to step back and stick to the skeleton representation, that allowed me to record a 50 seconds clip to show the performance.
- This way, the procedure have worked also for 15 minutes in a row, proving it way more stable than the other.

Conclusions

It would have been great to be able to display the body also in the realtime approach.

Conclusions

It would have been great to be able to display the body also in the realtime approach.
But, since it wasn't the main goal of the project, I decided to give up.
(Just because of this... I'm still slightly annoyed by that).

Conclusions

It would have been great to be able to display the body also in the realtime approach.
But, since it wasn't the main goal of the project, I decided to give up.
(Just because of this... I'm still slightly annoyed by that).
If I will ever have the chance, I would like to use this project as a starting point for some further experiments like, for example, trying to create a set of poses and animations for a character, or trying to use the realtime pose to make a comparison between the actual pose and another given pose.
Also, I'd like to test the **hand state** data to simulate something like the grasp of an object and moving it around.

Notes

- [1] *https://it.mathworks.com/matlabcentral/fileexchange/53439-kinect-2-interface-for-matlab*
- [2] *https://github.com/freaky1310/Kinect_3D_avatar*