Inverse Kinematics and Smoke Particles

Grant Smith [*]
Edinburgh Napier University
Physics-Based Animation (SET09119)

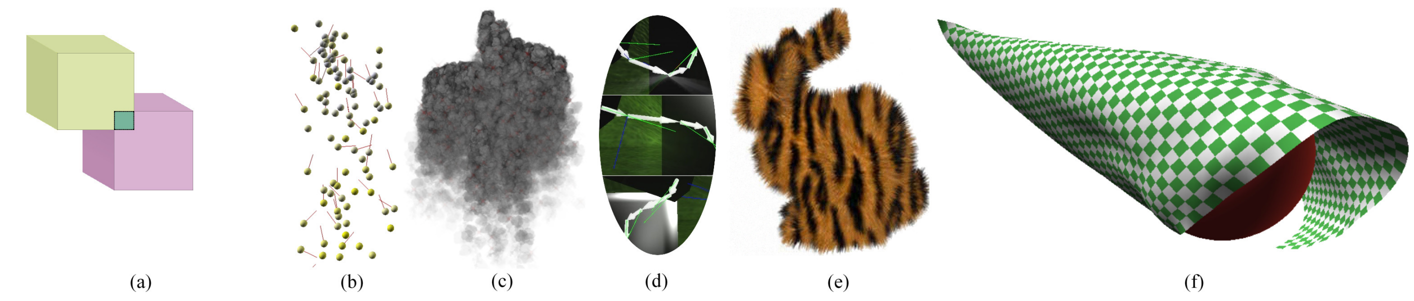

Figure 1: *(a) Collision Detection, (b) Particles, (c) Smoke Particles, (d) Inverse Kinematics, (e) Fur shells, and (f) position-based dynamics for cloth effects*

Abstract

The goal of this project was to create a scene demonstrating physics-based animation techniques. To achieve this, such techniques were researched before a final decision on what the project would include was decided upon. After research, Inverse Kinematics and Smoke Particles were chosen as the primary focus to demonstrate methods of creating animation through the use of physics. Alongside these, there needed to be post processing effects that would make the scene look more aesthetically pleasing and so, shaders were used to create the post processing effects such as screen space ambient occlusions (SSAO), Motion blur and Lens Flare.

Keywords: inverse kinematics, real-time, Smoke, Particles

1 Introduction

The aim of this project was to create a realistic looking scene using physics based animations, specifically inverse kinematics and smoke particles. The problem with creating a realistic looking scene with physics is that it is very easy to lose that realism due to there being so many tells, even if they are minor, that can completely break the sense of realism. For inverse kinematics and smoke particles, this is even more apparent. With inverse kinematics, it can be something as simple as an arm reaching for a button. There are many possible ways that the arm could reach the button, many are acceptable but there are also usually even more that just don't look right. In some cases, they even look unnatural (Compare Figure 2 and Figure 3). For smoke particles, the main problems are transparency and managing large sums of particles. Many particles are needed to create realistic smoke, which can be computationally expensive, but if there is a problem with transparency, no matter how many particles there are, the smoke will still look odd. Due to particles systems requiring large numbers of particles to look nice, it becomes difficulat to maintaing a steady frame-rate.

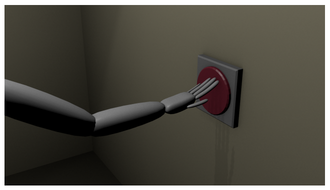

Figure 2: *Natural Looking Inverse Kinematics*

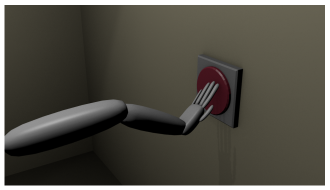

Figure 3: *Unnatural Looking Inverse Kinematics*

This simulation took inspiration from a few sources, some related to games and others to animation. Nvidia Gameworks and Unreal engine were the core motivators for smoke particles since both of them create realistic looking smoke and in the case of Nvidia Gameworks, the smoke particles can even be interacted with (See Figure 4) by the player through their actions such as moving the playable character through the smoke. This simulation includes interactive smoke and an inverse kinematics hierarchy that allows for an interesting, interactive scene that the user can explore. The techniques used have been optimized with real-time rendering in mind. Because of this, both the smoke and inverse kinematics could be used within a game to add realism and in the case of the inverse kinematics, it could reduce animating time by allowing certain animations to be generated on the fly using an end point rather than having an animator creating them.

[*]e-mail: 40111906@live.napier.ac.uk blog: migiesmith.blogspot.co.uk

Figure 4: *Smoke from Nvidia Gameworks*

Challenging & Limitations Inverse Kinematics and Smoke Particles can cause some difficulty when it comes to realism. It isn't hard to create a simple method for either but it can be problematic trying to achieve realistic results. For example, with inverse kinematics, the system may be able to find a way to reach the end point but it may reach it in a way that would appear unnatural (Compare Figure 2 and Figure 3) or even impossible in the real world. Smoke particles on the other hand can not only have issues with looking realistic but they can also become very intensive as the number of particles used grows. Finding a way to manage many particles while maintaining a steady frame rate requires a fair bit of extra work compared to just firing a small, limited amount of particles from an emitter.

Our Work The major limitation to this simulation is the frame-rate. Only so much data can be worked on and displayed on the screen before the frame-rate starts to plummet. Because of this, optimization has been taken into account to minimize the time taken to calculate or handle the data required by the systems covered withing this report (e.g. Particle Systems and Inverse Kinematics). With the particle system, optimizations such as frustum culling and other methods of adding and removing particles have been implemented.

Outline This report will cover work related to the features of the simulation in Section 2, followed by an overview of the simulation in Section 3. In Section 4, the simulation and it's components will be covered in detail including the aim of each component along with how it was implemented and the limitations of each. Section 5 describes the risks, the major software development tasks of simulation and the external libraries used. Finally in Section 6, the report will be brought to a conclusion

Starting examples: This report attempts to address the problem of using Inverse Kinematics in real-time rendering by considering the use of multiple existing methods. Additionally, this report attempts to address this problem when collision detection and smoke particles are thrown into the mix as well.

The report attempts to introduce the reader to the cross over between the world of computing and robotics by demonstrating methods of calculating inverse kinematic solutions to a hierarchy of links. Along side this, particle systems (mainly those consisting of smoke) and collision detection will be covered showing a relationship that these can have with each other through visual representations of their interactions and descriptions of how the systems work separately and together.

This report addresses the problem that occurs when combining effects such as inverse kinematics and particle system which includes the hit that is taken to the frame-rate of the simulation. This is addressed through the use of optimization techniques to minimize the worked required per update to achieve the resulting simulation.

2 Related Work

'FABRIK: A fast, iterative solver for the Inverse Kinematics problem' written by [Aristidou and Lasenby 2011] covers their fast, realistic inverse kinematics solver in comparison to other methods previously used. The comparison shows a clear improvement relative to previous methods such as Cyclic Coordinate Descent (CCD) and The Jacobian. FABRIK takes a leap forward in terms of making inverse kinematics a more viable solution to animation in games due to how much quicker it is compared to the previously stated methods. CCD takes roughly as many iterations to calculate the orientations of each link as FABRIK does but FABRIK computes this in just under a tenth of the time (these statements are going by the results written in this article [Aristidou and Lasenby 2011]). Unlike the CCD method, FABRIK calculates a much more realistic combination of rotations for the links making it stand out in comparison to The Jacobian which is slower than both FABRIK and CCD but is a lot more accurate than CCD. Of course though, with the improvements to both accuracy and speed, the FABRIK method is more complex than CCD, making it harder to implement, but with the the improvements that come with implementing it, that increased complexity is definitely worth the results.

Particle Systems have been used for many years to create effects that require a lot of components with limited durations. The uses can vary from small, quick effects like sparks or splashes to effects that are larger and last longer such as fire and smoke. 'Building an Advanced Particle System' by [Van Der Burg 2000] expands on the idea of a simple particle system to make it more advanced and diverse, the same system can be used for multiple types of particles (e.g. smoke, rain and sparks). The article covers the creation of an advanced particles system, including the procedures that the system follows and the data structures it requires to run. Optimization methods are also mentioned which are necessary for a particle system to function effectively without bringing the frame rate to the ground. One potential optimization that isn't brought up, which is likely due to when the article was created, is the use of the geometry shader to generate the triangles for the particles on the GPU. By doing so, the CPU wouldn't have to shuffle so much memory to handle the particles, no triangles would need to be generated on the CPU, they would all be generated from a single point by the GPU (which would receive the points from the CPU).

3 Overview

This simulation will make use of smoke particles and inverse kinematics to create animations using physics that are aesthetically pleasing while appearing realistic in the way that they move. The inverse kinematics used in this simulation will make use of angle limits and collision detection to prevent scenarios that would be deemed unrealistic in comparison to the real world. Angle limits will prevent joints from bending in ways that would be unnatural (Compare Figure 2 and Figure 3), which would break the realism that the simulation aims to achieve. Collision detection will be used for the inverse kinematics, smoke particles and in other situations within the simulation. The main purpose of the collision detection system will be to ensure that objects react correctly to contact with other objects (e.g. to prevent intersection of two objects or to make sure that one object doesn't just simply pass through another).

4 Simulation

4.1 Inverse Kinematics:

The inverse kinematics (See Figure 1(d)) component of the simulation is made up of two parts. These two parts are the methods used to calculate the inverse kinematics. The two methods are Cyclic Coordinate Descent (CCD) and The Jacobian. The difference between these two methods besides the way in which they work is the speed and the accuracy of the links position and rotation. With the jacobian method, solving the inverse kinematic problem is more CPU intensive but tends to find a better solution than the CCD method which is less CPU intensive.

This component of the simulation works by iteratively rotating links within the inverse kinematic hierarchy towards a target until the target is reached by the end point of the links.

Through the use of the keyboard and mouse, the user will be able to manipulate the simulation. It could be manipulated in ways such as dragging the end points of the inverse kinematics hierarchy around to manipulate the pose that the hierarchy, or part of it, is in. The user will also be able to toggle which inverse kinematic solver is being used to calculate the positions and on top of that, it will be possible for the user to add particle emitters to the scene on top of the ones that will already be there.

The main objective of the inverse kinematic portion of this simulation is to create a multi-end point hierarchy of links that can be used to animate a 3D object (or objects). This hierarchy must be able to a realistically and efficiently animate the 3D object (or objects) using just end points for the end links of the hierarchy to follow. This simulation must also allow for the method used to calculate the inverse kinematics of the hierarchy to be changed when the user chooses. The methods used will be the Cyclic Coordinate Descent (CCD) method and The Jacobian method. The inverse kinematics hierarchy will also be used to create structures representing the limbs of a human and other non-bipedal hierarchies. The human hierarchy will be implemented so that it is easy to determine if the inverse kinematics calculates paths to the target that appear realistic.

The basic premise for the CCD method is to iteratively rotate the links until the target is reached. To do so, each link calculates the axis it needs to rotate around and the angle it needs to rotate by. The axis is calculated by taking the cross product (See Equation 1) of the directional vector between the current link and the end link against the directional vector between the current link and the target. The angle the link needs to rotate by is then calculated by taking the dot product (See Equation 2) of the two directional vectors used in the cross product equation (Equation 1). The axis and angle are then used to create a quaternion that represents the rotation required to rotate the link from it's current rotation to it's desired rotation. This is then combined with the current rotation of the link to get that desired rotation. The process is then repeated for each link until an optimal, or close enough, solution is found. Each link's rotation and position is combined with the rotation and position of it's parent link (the hierarchical part of the system). A visual representation of the process can be see in Figure 5. A pseudo code version of the CCD reach algorithm explaining the process in detail can be seen in Algorithm 2 along with Algorithm 1 which covers the updating of the hierarchy.

$$axis = cross(currToEnd, currToTarget) \tag{1}$$

$$\begin{aligned} angle &= cross(currToEnd, currToTarget) \\ angle &= acos(angle) \end{aligned} \tag{2}$$

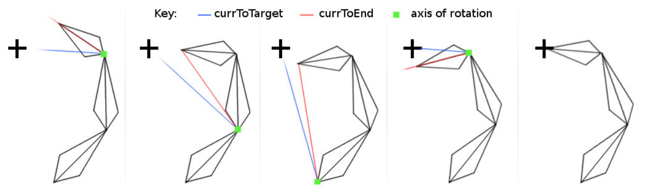

Figure 5: *Visual representation of the CCD process*

Algorithm 1 Hierarchy Update

```
 1: $rotation$ = convert $qOri$ to a 4x4 matrix
 2: transpose $rotation$
 3:
 4: $translation$ = translation matrix from $origin$
 5: if this link has a parent then
 6:     translate $translationby$ a vector with the x value of the parent's length
 7: end if
 8: $base = T1 * R1$
 9:
10: $qWorld = qOri$
11:
12: if this link has a parent then
13:     $m_base = parent_base * m_base$
14:     $qWorld = qOri * parent_qWorld$
15: end if
16: normalize $qWorld$
17:
18: for Each child $c$ do
19:     update $c$
20: end for
```

The inverse kinematics can be tested to prove it's effectiveness through the use of moving endpoints. These points could be rotated around another origin point to show that the hierarchy can follow them no matter where the points are around it. If the inverse kinematics hierarchy can reach the points (or point towards them if they are too far away) then the test is deemed a success since the hierarchy will be able to reach any other points given to it. This is true because this test will check all around the hierarchy, ensuring that it can in fact reach anywhere, since the points will be rotating 360 degrees on every axis (X, Y and Z).

The inverse kinematics hierarchy can be any size as long as the CPU can handle the the number of calculations needed to move the links towards the target. This scalability comes from the fact that each links can be added to the system in any order and quantity. These links could be added at any point, even at the end of another link that already has another link connected to that end (multiple child nodes), allowing for multiple end points and complex hierarchies.

The main limit for the inverse kinematics is the number of links. Only so many can be updated during each update loop due to the limitations that the CPU itself has (how many operations per second it can do). Because of this, there are no huge inverse kinematic hierarchies within the simulation as this would cause a large amount of lag which would inevitably make it hard to see the simulation and it would no longer be real-time.

4.2 Collision Detection:

The Collision Detection (See Figure 1(a)) used within the simulation consists of multiple methods of determining if two objects intersect. Each method varies in it's accuracy and performance, for example sphere v sphere is very fast in comparison to OBB v OBB

Algorithm 2 CCD Reach

```
$endVec$ = the end position of the end link
$currPos$ = transtlation from current link's transform matrix
$currToEnd$ = directional vector from $endVec$ and $currPos$
$currToTarget$ = directional vector from $target$ and $currPos$

$axis$ = cross productor of $currToEnd$ and $currToTarget$

if the magnitude of axis ¡ 0.01 then return
end if
normalize $axis$;

$angle$ = dot product of $currToEnd$ and $currToTarget$
clamp $angle$ between -1.0 and 1.0
$angle$ = inverse cos of $angle$

if $angle$ equals 0.0  then return
end if

$qDif$ = create normalized quaternion from -$angle$ and $axis$

$qCur$ = current link's rotation in world space;
normalize $qCur$

$qPar$ = identity quaternion;
if ( thenthis link has a parent)
    $qPar$ = parent's rotation in world space;
    normalize $qPar$
end if

$qLocal$ = $qCur$ * conjugate of $qPar$
$qNew$ = ($qCur$ * $qDif$) * conjugate of $qPar$
normalize $qNew$

slerp $qOri$ to $qNew$ by a factor of $physicsTimeStep$
normalize $qOri$

for Each child $c$ do
    reach $c$
end for

```

$target$ is a vector passed in, representing the position we are attempting to reach
$physicsTimeStep$ is the fixed time passed that the simulation uses
$qOri$ is the rotation of the link
$qLocal$ is the rotation of the link in local space
$qPar$ is the parent's rotation in world space
$qNew$ is the calculated new rotation for the link

since the process of sphere v sphere is a lot simpler. The methods used are:

1. Sphere v Sphere
2. AABB v AABB (Axis Aligned Bounding Box)
3. OBB v OBB (Orientated Bounding Box)
4. Combinations of types (e.g. sphere v OBB)

In it's simplest form, collision detection is about checking the properties of one object against the properties of another and checking if the results meet certain criteria. In the case of sphere v sphere the positions of each sphere are compared and checked against the sum of either sphere's radius. If the resulting value is less than or equal to zero then it is deemed that the two spheres do in fact collide/intersect.

Objectives Collision detection was implemented into the simulation with the prime purpose of improving the realism of the inverse kinematics and particle system aspects of the simulation. Collision detection would do this by allowing for components of each to determine if they were intersecting with other objects, enabling them to respond to collisions. This response would in turn prevent these sections of the simulation from achieving results that would be impossible in the real world (e.g. smoke that passes through a solid wall).

Procedures - Sphere v Sphere Sphere v Sphere collision detection (See Equation 3) is calculated by subtracting the position of one sphere (v1) from the position of the other (v0) resulting in a vector representing the distance between the two positions (mV). To get the distance between the two spheres, you then calculate the magnitude of the distance vector (mV). Lastly, to determine if the two spheres are intersecting, you subtract the sum of the radius of each sphere (r0 and r1) from the magnitude which returns the amount of intersection. If that amount is less than or equal to zero then the two spheres are intersecting, else they are not.

$$\begin{aligned} mV &= v0 - v1 \\ magnitude &= \sqrt{(mV_x)^2 + (mV_y)^2 + (mV_z)^2} \\ intersection &= magnitude - (r0 + r1) \end{aligned} \quad (3)$$

Procedures - AABB v AABB (Axis Aligned Bounding Box) AABB v AABB collision detection (See Algorithm 3) is done by comparing the x, y and z values of the positions (pos0 and pos1) and dimensions (dimen0 and dimen1) of the cubes against one another to determine if they overlap on each axis. This is done by comparing all of the x values of the positions and dimensions and then comparing all of the y, followed by all of the z. Effectively the intersection test for AABB v AABB is the same if statement repeated 3 times, just using a different element of the position and the dimension each time. This if statement is made up of two parts.The first part can be described as subtracting the position's x value of one cube (pos1) from the position's x of the other cube (pos0) and converting this to an absolute value (which is essentially the process of removing the negative sign if it's a negative number). The second is the addition of one cube's dimension's x value and the dimension's x value of the other cube. If the first part is less than the second then move onto the next if, replacing 'x' with the next element (y and then z on the next again). If this process succeeds at each stage (x, y and z) then the two cubes intersect.

Algorithm 3 AABB v AABB intersection test

```
1: if abs($pos0_x$ - $pos1_x$) ¡ $dimen0_x$ + $dimen1_x$ then
2:     if abs($pos0_y$ - $pos1_y$) ¡ $dimen0_y$ + $dimen1_y$ then
3:         if abs($pos0_z$ - $pos1_z$) ¡ $dimen0_z$ + $dimen1_z$ then
4:             cube0 intersects with cube1
5:         end if
6:     end if
7: end if
```

Procedures - OBB v OBB (Orientated Bounding Box) OBB v OBB intersection testing is calculated using multiple 1D intersection tests (See Figure 6). A 1D intersection test is where you project positions along an axis to get the position on a 1 dimensional line. You then use that position along with others calculated using the same method to determine if there is a separation between the objects along that axis. To start an OBB v OBB intersection test, you need to get the axes that you are going to project along. OBB v OBB requires 16 axes (See Algorithm 4) in total in order to determine with certainty that the two cubes do or do not intersect. The first 6 are the normals of the cubes. There are 12 normals in total but since half of them are just the opposite of another normal, you can ignore them. There are then another 9 axes which can be calculated by taking the cross product of each of the cube's normals against the other cube's normals(See lines starting with 'cross' in Algorithm 4). The 16th and final axis is calculated by getting the direction vector between the cubes by subtracting their positions and normalizing the result.

Once you have the axes, you then need to get the corners of each cube. That is done by taking the position of the cube and adding the dimensions (inverting certain axis each time as to get each corner) multiplied by the rotation to that position, repeating for each corner. This process is repeated once for each cube, resulting in two sets of eight corners

With the corners and axes, we can now start on the 1D intersection tests. For each axes, take the corners of both cubes and calculate the min and max values of each cube on that axis. The min and max values are calculated by taking the dot product of the current axis and the current corner, if the result is less than min, min becomes equal to the result and if the result is greater than max then max becomes equal to the result (See Algorithm 5 for a more detailed view on calculated the min and max). Once the min and max values are calculated, they are used in a 1D intersection test to determine if the two cubes cross over on that axis. A 1D intersection test (See Equation 4) is done by calculating the long and sum span using the min and max values and comparing the longSpan and sumSpan against one another. If the longSpan is less than or equal to the sumSpan, then there is an intersection on the axis. If on any axis it is found that there is no intersection, then the whole process can stop early since if there isn't an intersection on one axis then there is no overall intersection (intersection in 3D).

Algorithm 4 OBB v OBB axes

```
1: cube0.normals[0],
2: cube0.normals[1],
3: cube0.normals[2],
4: cube1.normals[0],
5: cube1.normals[1],
6: cube1.normals[2],
7: cross(cube0.normals[0], cube1.normals[0]),
8: cross(cube0.normals[0], cube1.normals[1]),
9: cross(cube0.normals[0], cube1.normals[2]),
10: cross(cube0.normals[1], cube1.normals[0]),
11: cross(cube0.normals[1], cube1.normals[1]),
12: cross(cube0.normals[1], cube1.normals[2]),
13: cross(cube0.normals[2], cube1.normals[0]),
14: cross(cube0.normals[2], cube1.normals[1]),
15: cross(cube0.normals[2], cube1.normals[2])
16: normalize(cube1.position - cube0.position)
```

Algorithm 5 Calculate min and max on axis

```
1: min0 = max float value;
2: max0 = min float value;
3: min1 = max float value;
4: max1 = min float value;
5: for Each i = 0 to 7 do
6:     aDist = dot(corners0[i], axis);
7:     if aDist ¡ min0 then min0 = aDist;
8:     end if
9:     if aDist ¿ max0 then max0 = aDist
10:    end if
11:    bDist = dot(corners1[i], axis);
12:    if bDist ¡ min1 then min0 = bDist;
13:    end if
14:    if bDist ¿ max1 then max0 = bDist
15:    end if
16: end for
```

$$\begin{aligned} longSpan &= max(max_0, max_1) - min(min_0, min_1) \\ sumSpan &= max_0 - min_0 + max_1 - min_1 \end{aligned} \tag{4}$$

max0 and min0 are the max and min values for one cube max1 and min1 are the max and min values for the other cube

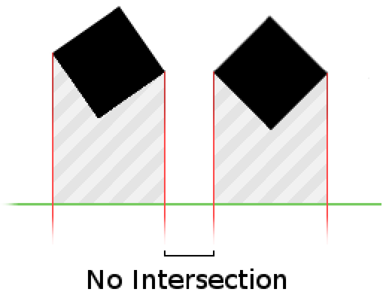

Figure 6: *1D intersection test*

Since collision detection isn't the main focus for this simulation, the methods used to detect the collisions have been restricted to those that are quite simple. This means that collision within the simulation will be checked using shapes such as spheres, cubes and planes. This limitation will allow other sections of the simulation to receive more CPU time, allowing for the other components to have fewer limitations. However, this limitation doesn't cause any issues since the other sections to not require there to be a high level of collision detection (e.g. convex collision detection).

5 Experimental Results

Major Software Development Tasks Particle Systems will likely be the largest issue when it comes to making them aesthetically pleasing while maintaining a steady frame-rate. The sheer amount of particles within the simulation will make it difficult to keep the simulation real-time in a way that isn't similar to a slide show. The likely solution to this issue will be to limit the number of particles quite drastically but methods of optimizing the particle systems will be researched and implemented in order to push it to the most ideal state, minimizing limitations where possible.

Risks The element of this simulation that will likely be the most difficult to implement will be the multiple methods of solving the inverse kinematics. This could be even more difficult than it would at first seems if the methods implemented conflict with the underlying systems already in place (e.g. using different maths libraries that have different ways of storing data like row major v column major matrices as an example).

External Libraries TinyXml [leethomason] - This was used to load in the mapping data for the text renderer. This data is then stored and used each time a character is rendered to the screen.

6 Conclusion

References

ARISTIDOU, A., AND LASENBY, J. 2011. Fabrik: a fast, iterative solver for the inverse kinematics problem. *Graphical Models 73*, 5, 243–260. 2

LEETHOMASON. Tinyxml. visited on 2015-10-18. 6

VAN DER BURG, J., 2000. Building an advanced particle system. 2