SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 3902

Procedural Skeletal Animation

Proceduralna animacija skeletnog modela

Bojan Lovrović

Zagreb, lipanj 2015.

CONTENTS

LIST OF FIGURES

List of Algorithms

1. Introduction

As demand for skeletal animation rises, alternatives to slow and tedious process of creating animation key frames manually have shown great results. One of the alternatives that produces realistic results faster is motion capture. Downside of this approach is that we end up having animation data like before. This is irrelevant if it is displayed just once in the content (e.g. video), but for applications in interactive content such as video games, where it can be displayed multiple times, it becomes very noticeable.

Procedural animation fixes this problem because it is different every time it is displayed and also, since it is computed automatically, removes the need for animator's time and effort. Application of procedural animation vary, but are mostly used to represent some physical behavior like particle systems (smoke, fire, water), cloth, rigid body dynamics, hair and fur, as well as skeletal animation. There are also other uses like turning video game character's head in the direction of the player when he/she is near.

Procedural skeletal animation can be viewed as an extension of rigid body dynamics and such approach of building one is presented in this paper. It is assumed there already exists a rigid body dynamics simulating engine and from there it will be explained what approaches were used ranging from modeling joint-like constraints, all the way to transforming from bone's local space to rigid body's and the other way around.

2. Modeling constraints

A few words should be said about the underlying physics engine. It is an *impulse-based* engine with *micro collisions* for collision resolution. What impulse based means is that when two bodies have a physical contact and closing velocity, an impulse is applied (as opposed to force) to remove it and perhaps add a certain amount of velocity in opposite direction based on the coefficient of restitution[1]. Micro collisions is a way the collision detection system works. It can only detect interpenetration once it has occurred. This is often called a discrete, static or a posteriori collision detection. The other way to implement such system is more expensive but provides extra precision. Such systems are called continuous, dynamic or a priori collision detectors and they take into account whole range of motion body will traverse in a time step. This results in no interpenetrations at all. Basic implementation of such system is shown in [2], as for implementation used here [5] is the best source of information. Uniform grid has been used for broad phase and separating axis theorem for narrow phase collision detection. Both of those algorithms can be found in [3].

There are two types of constraints used in presented implementation, hinge and universal joint. They both use the same constraint violation resolution system as shown in algorithm 1. It was done in such way to be compatible with existing collision detection resolver and since it uses depth of interpenetration as a value on which it bases it's search for most severe collision, an analogous value was required for joints. This is where *joint severity* value is calculated. Both joints have their own implementation for calculating this value, but it is fairly similar, as it will be shown in sections 2.1 and 2.2.

The algorithm 1 is based of the one described in [5]. Everything from the line 6. down is updating the severity of all the joints that share a rigid body with the currently updated joint. Each joint has pointers to rigid body instances whereas one body is a child and the second one is a parent. Other joint data varies on concrete implementa-

[1] A coefficient that affects the amount of separating velocity two objects will have after they collide.

Algorithm 1 Resolve constraint violation

```
1:  procedure RESOLVEALL(joints, iterationsNum)
2:      calculateSeverity(joints)
3:      for i ← 0 ... iterationsNum do
4:          joint = findMostSevere(joints)
5:          resolve(joint)
6:          parent ← getParent(joint)
7:          child ← getChild(joint)
8:          updateJoints ← ∅
9:          for j in joints do
10:             if j == parent or j == child then
11:                 updateJoints ← updateJoints ∪ j
12:         calculateSeverity(updateJoints)
```

tion.

Math behind *resolve* function is overly complex to be fully covered here and has books and papers based on this subject alone. There are, however, a few things worth mentioning here for better understanding of the given results. The translation that needs to be done to correct the position error is applied to each body in the amount reciprocal to their mass. In other words, a body with greater mass than the other member of the pair will move less to resolve the error. A similar approach is made for rotation, but with inertia tensor used instead of mass.

2.1. Hinge joint

Hinge joint is used to simulate behavior of a knee or elbow amongst other parts of skeletal system. It has just one degree of freedom which allows it to preform only two types of movement: flexion and extension. It can be said that this movement can define a *circle*. There are four values that describe given implementation. Two vectors: *axis* and *anchor* and two scalars: *back* and *front angle*. Axis represents the direction in which the rigid bodies will be able to rotate one relative to another. Anchor is a point about which rotation will occur. Both axis and anchor are saved in parent's and child's local space. Back and front angle are limits for rotation in both directions.

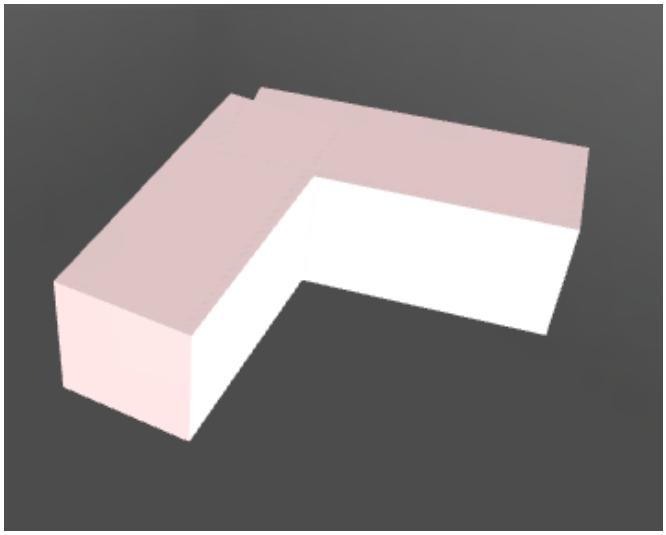

Figure 2.1: Hinge joint between two rigid bodies

Estimating hinge joint severity comes down to adding up four separately calculated values. *Position error* tells how much has a child body moved it's anchor point from it's parent's anchor point. Therefore it is also the amount of translation needed to correct this error.

Sideways bend error is a value representing the amount of transformation needed to return the bodies back on the circle previously defined. Equation 2.1 shows how to get an angle between current direction and the one on the circle. All values are in world space:

$$p = \mathbf{d} \cdot \mathbf{a} \tag{2.1}$$

$$\mathbf{d}' = \frac{\mathbf{d} - p \cdot \mathbf{a}}{\|\mathbf{d} - p \cdot \mathbf{a}\|} \tag{2.2}$$

$$s = acos(\mathbf{d}' \cdot \mathbf{d}) \tag{2.3}$$

Where **d** is normalized direction of a child rigid body and **a** is normalized joint axis. Since severity is actually estimated position error, angle can't be added to that sum. When working with small angles, radius can be multiplied with it to get the translation of the body, but whose radius, parent's or child's. The solution used here multiplies only a fraction of total angle with the radius of a body and the rest with the radius of the other one. How much rotation each body will get depends on their moments of inertia. The resulting value represents translation and can be added to severity sum.

As there can't be any rotation in the child's direction[2], *orientation error* gives the amount of rotation needed to apply to both bodies to correct this error if it exists. This value is computed easily as it comes down to working out an angle between axis from parent's and the one from child's local space when they are both transformed into world space. Angle is then used to get translation in a way analogous to that of the sideways bend error.

Lastly the *bend error* is used to determine how much did bodes violate the constraint put by previously defined back and front angle values.

Listing 2.1: Severity in hinge joint consists of four separately calculated values.

```
float HingeJoint::GetPosErrorSeverity()
{
  return mPosErrorSeverity + mSidewaysBendErrorSeverity +
    mOriErrorSeverity + mFEBendErrorSeverity;
}
```

2.2. Universal joint

Actually the model used here is something between universal joint (with its constraints) and ball-and-socket joint (with its additional degree of freedom). This results in a joint that has three degrees of freedom and therefore allows its rigid body pair to rotate in any arbitrary direction (up to a certain limit). Purpose for such constraint is application in skeletal systems that require joints such as shoulder and hip. Three values are required to define it. The *anchor* vector like in 2.1, the *rotational freedom* which sets boundary on rotation in the body's direction and *bend freedom* that limits rotation in the direction perpendicular to both bodies direction.

[2]Body's direction is defined as a vector going from it's anchor towards its center.

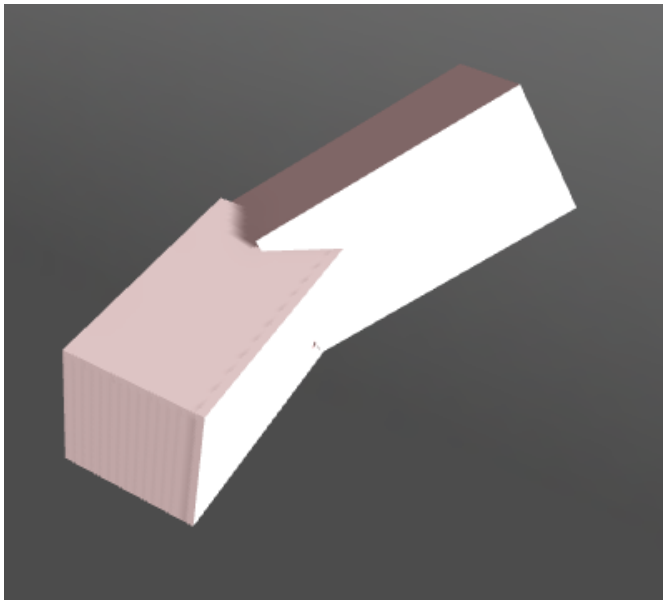

Figure 2.2: Universal joint between two rigid bodies

Method for estimating severity is almost the same as it is for hinge joint. This time only three values are taken into account. *Position error*, *orientation* and *bend error* are the ones left and they have the same meaning as they did in 2.1.

3. Skeletal animation

This chapter will move away from physics and simulations a bit as it covers implementation of the animation handling system. Used file format will be introduced first, following with the algorithm implemented for keyframe[1] selection. Methods presented here are based on [4].

3.1. Loading the animation data

File format that was used in this implementation was required to have exporters, in widely used 3D modeling programs such as 3DS Max. It also needed to have some sort of publicly accessible reference information, so that it is possible to write a parser for it. Both of those requirements were satisfied by choosing DirectX (.x) file format [1].

Listing 3.1: Data structure made to save vertex information.

```
struct SkinnedVertexFull {
  XMFLOAT3 Position;
  XMFLOAT3 Normal;
  XMFLOAT3 TangentU;
  XMFLOAT2 TexC;
  XMFLOAT3 Weights;
  BYTE BoneIndices[4];
};
```

Each vertex on animating mesh (usually called the *skinned mesh*) has values shown in listing 3.1. *BoneIndices* represents four indices of four bones that influence the vertex. As of how much exactly do they influence it, that information is stored in *Weights*. Note that it only has three floating point values. This is due to fact that all

[1] A typical data structure used for animation. Each keyframe is defined with time and some positioning value (e.g. position and orientation).

four bone weights must sum up to 1, thus making it easy for fourth component to be calculated in the vertex shader. Reason for this complication is that data in listing 3.1 must be sent to the graphics device and since this transition is a bottleneck of the whole system, the amount of data should be minimized.

Listing 3.2: Structures used for loading and rendering animation data.

```
struct Bone {
  // Name of the bone
  std::string mName;
  // Index of the bone's parent (root has -1)
  int mParent;
  // Transforms from world to bind space of a bone
  XMFLOAT4X4 mBind;
};
struct AnimationKey {
  // Animation time in time stamps
  UINT mT;
  // Transformation matrix
  XMFLOAT4X4 mW;
};
struct Animation {
  // Index of a bone this animation is for
  UINT mBone;
  // All animation keys for this bone and this animation
  std::vector<AnimationKey> mAnimKeys;
};
// All the data for one animation
struct AnimationSet {
  // name of the animation
  std::string mName;
  // Animations for specific bones in the skeleton that,
  // when combined, create animation set.
  std::vector<Animation> mAnim;
  // Length (in in time stamps) of the longest animation.
  UINT mTotalTime;
};
```

Listing 3.2 should be pretty much self explanatory with just one exception that should be elaborated. Variable named *mBind* is said to represent a transformation from world to bind space. Reason for this is the fact that the vertices influenced by a bone are not relative to the coordinate system of the bone (they are relative to the bind space[2], which is in the coordinate system the mesh is modeled in). So before transforming certain vertices from bone's local space to world space, we first need to transform the vertices from bind space to the space of the bone. This is often called *offset transformation* and those matrices must be provided in the file. Described method can be seen on figure 3.1.

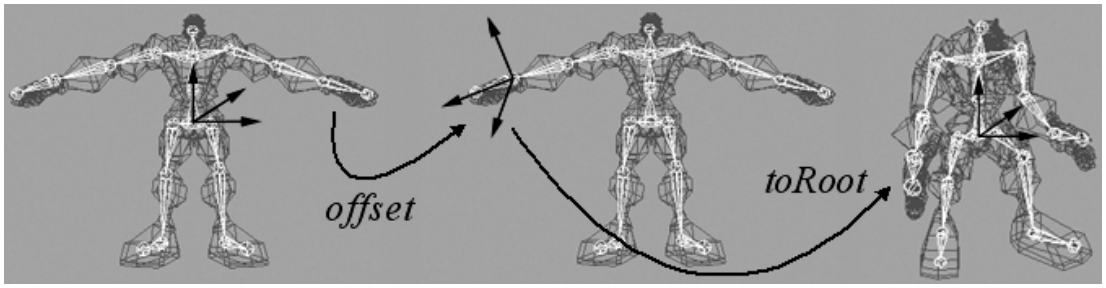

Figure 3.1: The order in which each vertex is multiplied to be transformed into final (animated) position.

3.2. Rendering the animation

In order to get the transformation from bone system to the world equation 3.1 is used recursively.

$$toRoot_i = toParent_i \cdot toRoot_p \tag{3.1}$$

Combined with method shown in section 3.1 and in equation 3.2

$$final_i = offset_i \cdot toRoot_i \tag{3.2}$$

results in algorithm 2.

The only matrix in algorithm 2 that is updated before every frame is the *toParent[]*. Linear interpolation is used to acquire translations in between keyframes and spherical linear interpolation is used on quaternions. If any scaling data is contained within the

[2]Bind space of a bone is a matrix created when the mesh was binded to a skeletal system.

Algorithm 2 Compute Final Bone Transform

1: **procedure** SETFINALTRANSFORM($boneIndex, parentToRoot$)
2: $toRoot = matrixMul(toParent[boneIndex], parentToRoot)$
3: $final = matrixMul(offset[boneIndex], toRoot)$
4: **for** $i \leftarrow 0$ ... $(numBones - 1)$ **do**
5: **if** $isChild(i, boneIndex)$ **then**
6: $SETFINALTRANSFORM(i, toRoot)$
7: **procedure** COMPUTEBONETRANSFORMS
8: $I = matrixIdentity()$
9: $rootIndex = getRootBoneIndex()$
10: $SETFINALTRANSFORM(rootIndex, I)$

animation keyframe transformations it will be discarded, since it would only complicate the solution and bring very little in terms of features.

After all the bone final transform matrices have been calculated on the CPU, they are sent in an array-type structure to the graphics card. Vertex shader then takes only four of those matrices from the array for each vertex, multiplies them with the world transform of the mesh, and obtains four positions for said vertex. Then they are simply multiplied with the four weights, each representing the influence of its bone and added together to form a final position of the vertex.

Figure 3.2: Image of the Khnum monster in it's bind space (left) and while animating (right)

4. Skeletal simulation

The final part of this paper merges sections 2 and 3 to create a popular procedural skeletal animation called a *ragdoll*. It also explains some additional methods used to achieve a more convincing physically based behavior.

4.1. Saving the ragdoll

Although majority of ragdoll setup could be determined by the program (e.g. bone length, anchor position), there is a lot of detail that requires human input (e.g. bone thickness, joint type). Said input therefore needs to be saved. If there is no file describing the ragdoll setup, the program will then use its assessment of possible values and create one, which can be later tweaked by the user.

4.2. Binding rigid bodies to bones

In order to achieve realistic transition from animation to simulation, rigid bodies need to be transformed to the positions and rotations of their respective bones. Before that can be done, ragdoll needs to be initialized in bind position for joints to be created correctly and symmetrically. What follows is disabling collision detection between bodies in parent-child and siblings relations, so that joints have a complete control over motion of rigid body pairs. Lastly, rigid bodies are transformed to final positions using the same *final* matrices from section 3.2.

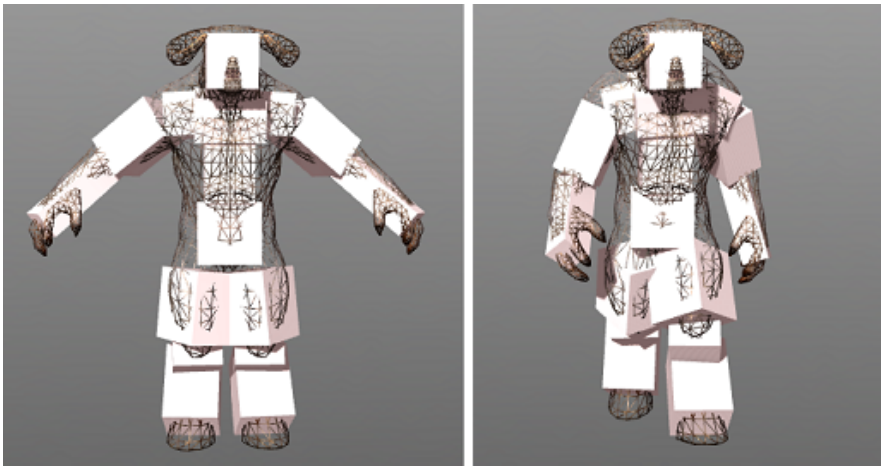

Figure 4.1: Binding rigid bodies to bones

The process can bee seen in figure 4.1. In the beginning of the simulation skinned mesh is transformed into bind position, where the rigid bodies and joints are being created (image on the left). Right after that (without rendering) whole mesh and all the rigid bodies are transformed back to the position in which the animation was interrupted (image on the right). Note that image (frame) on the left is never actually displayed.

4.3. Binding bones to rigid bodies

Once the simulation has started and rigid bodies have been transformed to their's respective bone's space, they are updated each frame and their new position is being applied to the bone they are bind to. What is meant by that is simply setting the value of bone's final transform matrix to that of a rigid body's current transformation. Before bone can be transformed by it's rigid body's transformation, it first must be transformed into given rigid body's local coordinates. This matrix will be conveniently called *boneInRigidBodySpace* and is given by equation 4.1.

$$
\begin{gathered}
boneBindToWorld_i = offset_i^{-1} \\
worldToRigidBodyBind_i = rigidBodyTransformBind_i^{-1} \\
boneInRigidBodySpace_i = boneBindToWorld_i \cdot worldToRigidBodyBind_i
\end{gathered}
\tag{4.1}
$$

Where *offset* is discussed in section 3.1 and *rigidBodyTransformBind* is simply a matrix that transforms rigid body to world coordinates in a binding position (left image on

figure 4.1). Note that this matrices are being calculated right after the model has been loaded, since only bind space information is required and those are constants provided by the model file.

With that out of the way, calculating bone final transform is straightforward and is given by:

$$worldToRigidBody_i = rigidBodyTransform_i^{-1}$$

$$boneFinal_i = boneInRigidBodySpace_i \cdot worldToRigidBody_i \cdot world^{-1} \quad (4.2)$$

Where *rigidBodyTransform* transforms rigid body to world space in current simulation position. The reason behind multiplying everything with inverse world transform is in that after applying *worldToRigidBody* vertices are already in world space and as described in section 3.2, everything is transformed by the world matrix in the shader. Naturally transforming something from world space to world space doesn't make sense and results in errors. Therefore putting the inverse world transform in the end of equation 4.2 cancels the world transform in the shader.

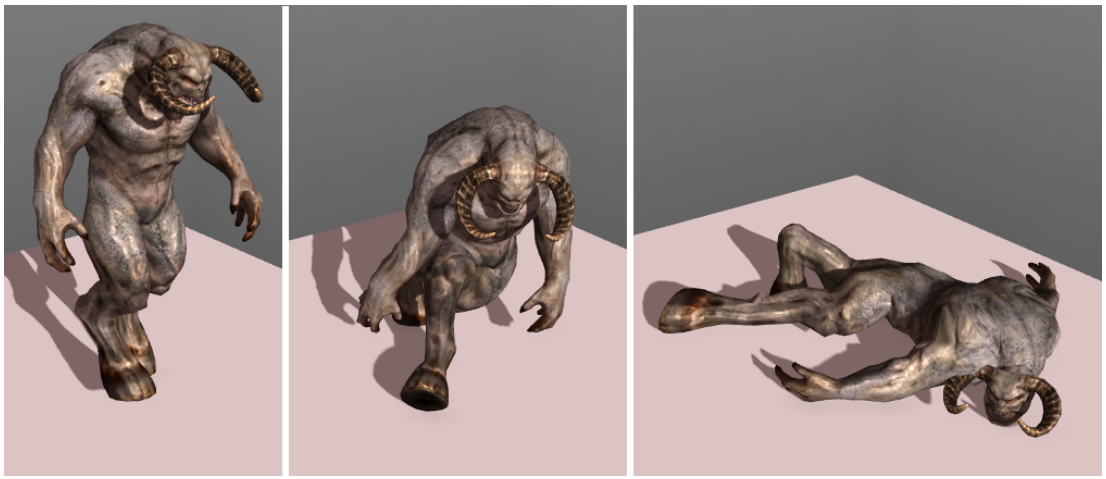

Figure 4.2: Transition from animation to simulation (left to right)

4.4. Further improvements

One part that is still missing and is really having a negative impact on how realistic simulation looks is kinetic energy of the bones. When an animation starts, each bone moves in it's own direction and therefore has it's own kinetic energy. This data is saved each frame using equation:

$$vel_i = \frac{p_i - p_{i-1}}{\Delta t} \tag{4.3}$$

Where *i* is a number of the frame, *p* is position and delta time is time passed from the last frame to the current one. This results in velocity for each bone in animation and when those velocities are passed to rigid bodies, kinetic energy is preserved. Note that only linear kinetic energy is being taken into consideration and rotational is being discarded. This approach, although not as precise, simplifies code and makes it run faster than it would if rotational kinetic energy was also calculated.

Method not implemented here but often used is damping. It could easily be one of the first next features to add to a system described here. Damping allows joints to try and slow down rigid body pair with excessive relative velocities. This results in a more realistic physical behavior since there is always friction present, and when simulating joints in humans and animals, there is an illusion of muscle providing resistance.

5. Conclusion

Although it can impact application's performance, procedural skeletal animation has proven, during the years, as a great way to decrease workload and improve realism. Consequently, it is a widespread practice to use it in animation and video game industry.

The results of this implementation were satisfactory, with the only downfall being the fact that user needs to tweak a text file in order to achieve better simulation, which is an easily fixable problem. Other than that, the simulation is providing infinite amounts of "animations" whenever needed, exactly what the purpose was.

BIBLIOGRAPHY

[1] https://msdn.microsoft.com/en-us/library/windows/desktop/bb173015(v=vs.85).aspx.

[2] David H. Eberly. *Game Physics (Second Edition)*. 2010.

[3] Christer Ericson. *Real-Time Collision Detection*. 2005.

[4] Frank D. Luna. *Introduction to 3D Game Programming with Directx 11*. 2012.

[5] Ian Millington. *Game Physics Engine Development*. 2007.

Proceduralna animacija skeletnog modela

Sažetak

Ovaj rad objedinjuje sustav animacija sa sustavnom za simulaciju gibanja krutog tijela u cjelinu koja omogućava proceduranu animaciju skeletnog modela. Obrađen je postupak izrade zglobova koji služe za spajanje krutih tijela, objašnjen je postupak animacije skeletno modela, te predočene su strukture podataka koje su korištene. Na kraju je predstavljen način na koji je implementirano konačno riješenje te dodatno metode za dobivanje na kvaliteti simulacije.

Ključne riječi: skeletalni model, fizikalna simulacija, proceduralna animacija

Procedural Skeletal Animation

Abstract

This paper encapsulates animation system with rigid body dynamics in a unity that enables procedural skeletal animation. Procedure for creating joints that connect rigid bodies and procedure for skeletal model animation are described. Data structures used are also described. At the end, the final result is presented as well as the additional method for improving quality of the simulation.

Keywords: skeletal model, physics simulation, procedural animation, ragdoll