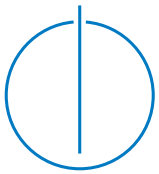
FAKULTÄT FÜR INFORMATIK

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatik: Games Engineering

Charakter-Animation mit der Bullet-Engine

Kwon-Jin Maximilian Jensen



FAKULTÄT FÜR INFORMATIK

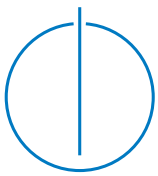
TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatik: Games Engineering

Charakter-Animation mit der Bullet-Engine

Character Animation using the Bullet engine

Author:	Kwon-Jin Maximilian Jensen
Aufgabensteller:	Prof. Dr. Rüdiger Westermann
Betreuer:	Prof. Dr. Rüdiger Westermann
Abgabedatum:	15.09.2015



Ich versichere, dass ich diese Bachelor's Thesis selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

München, 15.09.2015 Kwon-Jin Maximilian Jensen

Inhaltsverzeichnis

1 Einführung

Ziel dieses Projekts war die Implementierung eines Echtzeit-Charaktersystems auf Basis der Bullet-Engine. Es sollte möglichst viele von modernen Spiele-Engines erwartete Features aufweisen. Dazu gehören beispielsweise Ragdolls, Softbodies und inverse Kinematik. Dadurch sollte unter anderem die Tauglichkeit der Bullet-Engine überprüft werden. Das Programm wurde mit Hilfe von C++ und OpenGL implementiert.

Desweiteren wurden folgende Open-Source-Libraries verwendet:

- Open-Asset-Import-Libary: Zum Laden von fbx-Dateien
- SDL2: Zur Erstellung des OpenGL-Kontexts und für Input-Events
- Freetype: Für das Text-Rendering im selbstentwickelten GUI-Framework
- GLM: Für mathematische Funktionen

Als zusätzliche Lernmittel dienten:

- OGLdev Modern OpenGL Tutorials [Mei]
- Blenderella, Character Modeling in Blender 2.5 DVD [Gue]

2 Charakter-Erstellung

Der Charakter besteht hauptsächlich aus einem Skelett, einem Ragdoll-Modell und beliebig vielen Mesh-Komponenten. Diese müssen in einer festen Reihenfolge erstellt werden. Die Erstellung des Ragdoll-Modells erfolgt nämlich anhand der Bones und den ihnen zugewiesenen Vertices. Daher müssen das Skelett und alle Mesh-Komponenten, die bei der Berechnung der Rigidbodys mit einbezogen werden sollen, bereits geladen sein.

```
CharacterObject* testCharacter = new CharacterObject(...);
testCharacter->loadSkeleton("Data/Skeleton/characterBaseSkeleton.fbx");
testCharacter->addMeshComponent("Data/MeshComponents/characterBase.fbx");
testCharacter->createRagdoll();
```

Danach können die Mesh-Komponenten beliebig ausgetauscht werden. Das Ragdoll-Modell verändert sich dabei nicht mehr.

```
testCharacter->removeMeshComponent("Data/MeshComponents/characterBase.fbx
    ↪ ");

testCharacter->addMeshComponent("Data/MeshComponents/head.fbx");
testCharacter->addMeshComponent("Data/MeshComponents/arms.fbx");
testCharacter->addMeshComponent("Data/MeshComponents/shirt.fbx");
testCharacter->addMeshComponent("Data/MeshComponents/pants.fbx");
testCharacter->addMeshComponent("Data/MeshComponents/boots.fbx");
testCharacter->addClothComponent("Data/ClothComponents/cape.fbx");
```

2.1 Skelett

In der „loadSkeleton“-Methode werden mit Hilfe der Open-Asset-Import-Library das Skelett und alle dafür erstellten Keyframe-Animationen aus einer fbx-Datei geladen. Das Skelett besteht aus einer hierarchisch strukturierten Verkettung von „Bone3D“-Instanzen. Außerdem wird eine Map erstellt um über die Bone-Namen auf die Bones

zugreifen zu können. Zudem werden auch schon mal die später benötigten IK-Chains erstellt.

```
characterSkeleton->loadSkeleton(fileName);
leftFootIK = characterSkeleton->addIKChain("foot.L", 3);
rightFootIK = characterSkeleton->addIKChain("foot.R", 3);
```

2.2 Ragdoll

Für ein funktionsfähiges Ragdoll-Modells sind Rigidbodys und Constraints notwendig. Die Rigidbodys werden für ausgewählte Bones erstellt. Allerdings braucht nicht jeder Bone einen eigenen Rigidbody. Für die einzelnen Finger beispielsweise wurden keine Rigidbodys erstellt. Es muss lediglich sichergestellt werden, dass am Ende alle Rigidbodys durch Constraints miteinander verbunden sind.

All dies geschieht in der „createRagdoll“-Methode. Dabei werden die Rigidbodys mit einer Masse von 0 initialisiert, sie sind also zunächst statisch. Die Constraints bleiben ebenfalls anfangs noch deaktiviert. Der Grund hierfür ist, dass die Rigidbodys nur dann von der Bullet-Engine simuliert werden sollen, wenn sich der Charakter im Ragdoll-Modus befindet. Ansonsten werden sie bei jedem Update mit den entsprechenden Bones synchronisiert.

2.2.1 Rigid Bodies

Die Erstellung eines Rigidbodys für einen bestimmten Bone erfolgt durch einen einfachen Funktionsaufruf:

```
addRigidBody("upper_arm.L", 0.7f);
```

Diese Funktion sucht nach einem Bone mit dem Namen „upper_arm.L“ und allen Vertices die für diesen Bone eine Gewichtung von mindestens 0.7 haben. Für diese Vertices wird im Koordinatensystem des Bones eine Bounding Box erstellt. Die Ausmaße dieser Bounding Box werden für die Erstellung des Rigidbodys verwendet. In Abbildung 2.1 wird dies veranschaulicht.

2.2.2 Constraints

Als letztes müssen noch die Constraints hinzugefügt werden. Dazu werden die folgenden drei von der Bullet-Engine zur Verfügung gestellten Constraint-Typen verwendet:

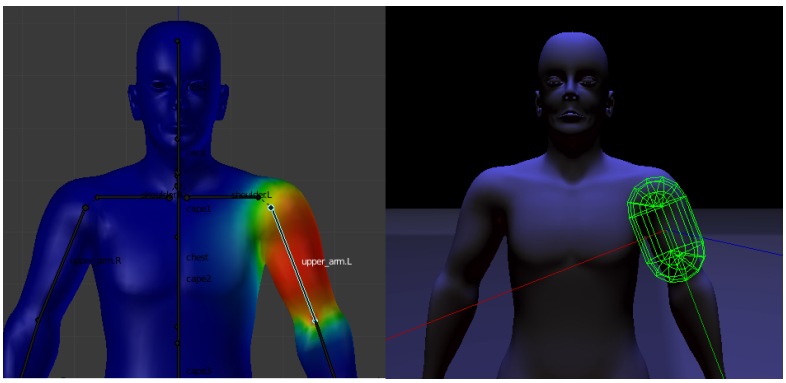


Abbildung 2.1: Vertex Weights in Blender und der daraus berechnete Rigidbody

„btGeneric6DofConstraint“, „btConeTwistConstraint“ und „btHingeConstraint“. Um beispielsweise die Rigidbodys der Bones „upper_arm.L“ und „forearm.L“ durch ein „btGeneric6DofConstraint“ miteinander zu verbinden wird der folgende Methoden-Aufruf getätigt:

```
characterSkeleton->add6DofConstraint(dynamicsWorld, "upper_arm.L", "
    ↪ forearm.L", glm::vec3(-3 * M_PI_4, 0, 0), glm::vec3(0, 0, 0));
```

Diese Methode überprüft als erstes ob Bones mit den Namen „upper_arm.L“ und „forearm.L“ überhaupt existieren und über Rigidbodys verfügen. Wenn ja, wird ein „btGeneric6DofConstraint“ mit den übergebenen Werten als unteres und oberes Limit für die Rotations-Freiheitsgrade erstellt.

3 Update-Logik

Das Update erfolgt in drei Schritten. Als erstes wird in der „CharacterController“-Klasse die grundsätzliche Position, Geschwindigkeit und Orientation des Charakters berechnet. Abhänging von diesen Werten wird dann in der „CharacterSkeleton“-Klasse die aktuelle Pose durch eine Kombination verschiedener Keyframe-Animationen berechnet. Abschließend wird diese Pose nochmal so verändert dass die IK-Constraints erfüllt werden.

3.1 CharacterController

An dieser Stelle wäre es auch möglich gewesen den von der Bullet-Engine zur Verfügung gestellten „btKinematicCharacterController“ zu verwenden. Für mehr Flexibilität wurde aber stattdessen eine eigene simple Klasse implementiert. Sie dient als Schnittstelle zur Steuerung des Charakters. Zu diesem Zweck wird ein kugelförmiger Bullet-Rigidbody als Repräsentation des Charakters verwendet (siehe Abbildung 3.2). Tastatureingaben bewirken die Anwendung von Kräften auf diesen Rigidbody. Die Steuerung erfolgt in typischer Third-Person-Manier durch eine Kombination von Maus und Tastatur. Dabei bestimmt die Maus die Blickrichtung, während die Pfeiltasten (oder wasd) eine Beschleunigung relativ zu dieser Richtung bewirken. Der Zustand der derzeitigen Tastatureingabe wird in den Variablen „walkingX“ und „walkingZ“ festgehalten. Diese sind Integer-Variablen welche die Werte -1, 0 und 1 annehmen können.

```
// -1: left, 0: no x-movement, 1: right
int walkingX;
// -1: backward, 0: no z-movement, 1: forward
int walkingZ;
```

Um die genaue Richtung der anzuwendenden Kräfte zu berechnen, müssen der Vorwärts- und der Rechtsvektor der Kamera projeziert auf die xz-Ebene miteinbezogen werden.(Siehe Abbildung 3.1)

Danach wird eine Kraft entlang dieser Richtung auf den Rigidbody angewendet. Damit die Kugel nicht weiterrollt nachdem die Tasten losgelassen wurden, wird eine

```
glm::vec3 direction(0, 0, 0);

if (walkingX != 0) {
    glm::vec3 directionX = camera->getRightVector() * (float)walkingX;
    directionX.y = 0;
    directionX = glm::normalize(directionX);
    direction += directionX;
}
if (walkingZ != 0) {
    glm::vec3 directionZ = camera->getDirectionVector() * (float)walkingZ;
    directionZ.y = 0;
    directionZ = glm::normalize(directionZ);
    direction += directionZ;
}
if (direction != glm::vec3(0, 0, 0))
    direction = glm::normalize(direction);
```

Abbildung 3.1: Richtungsberechnung der Kraft

der xz-Komponente des Geschwindigkeitsvektors entgegengesetzte Bremskraft angewendet. Außerdem muss sichergestellt werden, dass eine Maximalgeschwindigkeit nicht überschritten wird.

Wesentlich für die Darstellung des 3D-Modells sind die Position und der auf die xz-Ebene projezierte Geschwindigkeitsvektor des Rigidbodys. Dieser Vektor bestimmt nämlich die Blickrichtung des 3D-Modells und das Maß in dem die „Idle“-, „Walk“- und „Run“-Animationen zusammengemischt werden.

3.2 CharacterSkeleton

3.2.1 Animation Blending

Zunächst müssen die für das Animations-Blending benötigten Animationen bestimmt werden. Die zuvor berechnete Geschwindigkeit wird dazu mit den Konstanten 0, „walkSpeed“ und „runSpeed“ verglichen. Entsprechend kann die Enum-Variable „animationState“ die Werte „IDLE“, „IDLE_WALK“, „WALK_RUN“ oder „RUN“ annehmen.

Jedes Mal wenn „animationState“ neu gesetzt wird, müssen „AnimationBlendComponent“s neu erstellt werden. Jede „AnimationBlendComponent“ enthält die drei Variablen „animationName“, „animationStatus“ und „weight“. Die „weight“-Werte

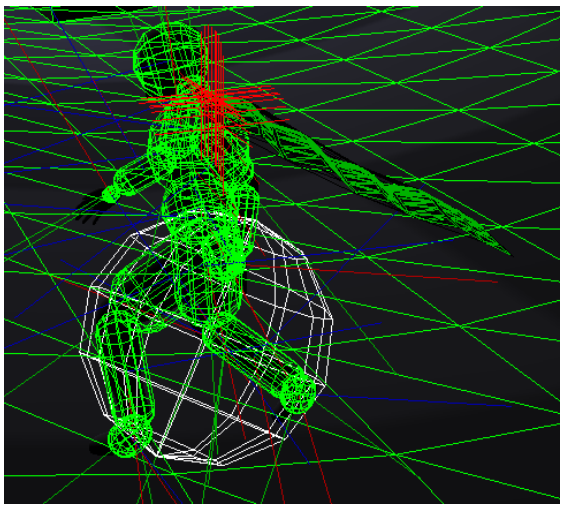


Abbildung 3.2: Debug Ansicht des Character-Controllers

sollten zusammen 1 ergeben. „AnimationStatus“ ist eine Float-Variable, welche Werte von 0 bis 1 annimmt und angibt wie weit die Animation fortgeschritten ist. Für „IDLE_WALK“ beispielsweise werden die „AnimationBlendComponent“s folgendermaßen erstellt:

```
else if (vel > 0 && vel <= walkSpeed && animationState != IDLE_WALK) {
        characterSkeleton->clearBlendComponents();
        idleComponentId = characterSkeleton->addBlendComponent("AnimStack::
            ↪ metarig|Idle", idleAnimationStatus, 0.5f);
        walkComponentId = characterSkeleton->addBlendComponent("AnimStack::
            ↪ metarig|Walk", walkRunAnimationStatus, 0.5f);
        animationState = IDLE_WALK;
}
```

Abbildung 3.3: Erstellung der Blend-Komponenten für die Animation

Die Gewichtungen von 0.5f sind dabei willkürlich gewählt, denn sie werden bei jedem Update neu berechnet. In Abbildung 3.5 sieht man, dass die „walk“- und „run“-Animationen synchron voranschreiten. Daher reicht eine „AnimationStatus“-Variable für beide Animationen. Ihr Inkrement wird mit der Geschwindigkeit und „deltaTime“ multipliziert. Dies bewirkt, dass die Abspielgeschwindigkeit der Bewegungsanimationen zur Geschwindigkeit des Charakters proportional ist. Die „idle“-Animation, welche das Atmen darstellt und nur im Stand und bei langsamen Gehen sichtbar

ist, soll dagegen immer mit konstanter Geschwindigkeit ablaufen und hängt daher nur von „deltaTime“ ab. Abbildung 3.4 zeigt an einem Beispiel wie das Update der Blend-Komponenten implementiert wurde.

```
idleAnimationStatus = fmod(idleAnimationStatus + 0.5f * deltaTime, 1.0f);
if (animationState == IDLE) {
        characterSkeleton->setBlendComponentStatus(idleComponentId,
            ↪ idleAnimationStatus);
}
else {
        walkRunAnimationStatus = fmod(walkRunAnimationStatus + 0.2f * vel
            ↪ * deltaTime, 1.0f);
        if (animationState == IDLE_WALK) {
                characterSkeleton->setBlendComponentWeight(idleComponentId,
                    ↪  1.0f - vel / walkSpeed);
                characterSkeleton->setBlendComponentWeight(walkComponentId,
                    ↪  vel / walkSpeed);
                characterSkeleton->setBlendComponentStatus(idleComponentId,
                    ↪  idleAnimationStatus);
                characterSkeleton->setBlendComponentStatus(walkComponentId,
                    ↪  walkRunAnimationStatus);
        }
        else if (animationState == WALK_RUN) {
                ...
        }
        else if (animationState == RUN) {
                ...
        }
}
```

Abbildung 3.4: Update der Blend-Komponenten

3.2.2 Forward Kinemetics

Im nächsten Schritt wird rekursiv für jedes Bone die Methode „updateBoneTransforms“ aufgerufen. Hier werden Skalierung, Rotation und Translation zu den in den „AnimationBlendComponent“s spezifizierten Zeitpunkten aus den Keyframe-Animationen herausgelesen und anhand ihrer Gewichtungen interpoliert. Abbildung 3.6 zeigt dies

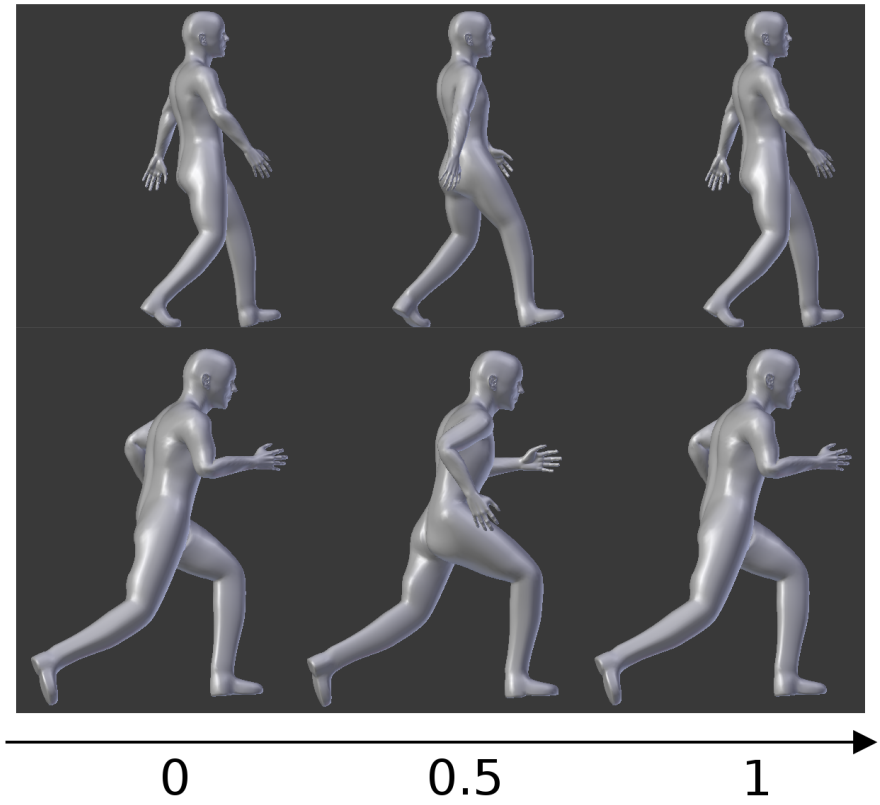


Abbildung 3.5: Vergleich der Walk- und Run-Animation

für die Rotation.

Am Ende der Funktion werden die relevanten Transformations-Matrizen, wie in Abbildung 3.7 dargestellt, gespeichert. Außerdem wird die Funktion rekursiv für alle Kind-Bones aufgerufen.

3.2.3 Inverse Kinematics

Um die Füße des Charakters besser auf unebenem Boden zu platzieren wird die berechnete Pose nun durch inverse Kinematik angepasst. Dazu wurde für jedes Bein eine IK-Chain erstellt, welche beim „thigh"-Bone beginnt und am „foot"-Bone endet. Die Füße sind also die Endeffektoren. Durch Raycasting werden die Zielpositionen bestimmt. Da sich die Endeffektoren zu Beginn unterhalb des Bodens befinden können, müssen die Startpositionen der Rays ein wenig nach oben verschoben werden. Außerdem müssen die Zielpositionen ebenfalls angepasst werden, denn sonst würden die Füße

beim Laufen am Boden kleben. Hierzu wird einfach die Höhendifferenz zwischen Endeffektor und Fußpunkt der Charaktercontroller-Kugel zur Zielposition addiert.

Das implementierte Verfahren zur Berechnung orientiert sich an einem von Buss [Bus09] veröffentlichten Artikel. Da sich die IK Chains im implementierten Fall nicht gegenseitig beeinflussen, können sie als seperate Problemstellungen mit jeweils einem Endeffektor betrachtet werden. Die Position dieses Endeffektors $\overrightarrow{\mathbf{s}}$ hängt von den Winkeln θ_i an den Joints ab. Es werden Winkel gesucht, so dass diese Position mit einer bestimmten Zielposition $\overrightarrow{\mathbf{t}}$ übereinstimmt. Es soll also gelten:

$$\overrightarrow{\mathbf{s}}(\overrightarrow{\theta}) = \overrightarrow{\mathbf{t}}$$

Durch ein iteratives Verfahren kann eine gute Lösung approximiert werden. Dazu wird die Jacobi-Matrix wie folgt aufgestellt:

$$\mathrm{J} = \left(\frac{\partial \overrightarrow{\mathbf{s}}}{\partial \theta_i} \right)_i$$

Für kleine $\Delta \overrightarrow{\mathbf{s}}$ gilt:

$$\Delta \overrightarrow{\mathbf{s}} \approx \mathrm{J} \Delta \overrightarrow{\theta}$$

Allerdings ist J im Allgemeinen nicht invertierbar. Eine performante Lösung, welche in diesem einfachen Fall bereits ein hinreichend gutes Ergebnis liefert, ist die Verwendung der Transponierten J^T statt J^{-1}. Für $\Delta \overrightarrow{\mathbf{s}}$ wird der Vektor $\overrightarrow{\mathbf{e}} = \overrightarrow{\mathbf{t}} - \overrightarrow{\mathbf{s}}$ skaliert mit dem Faktor α eingesetzt. Die Winkel $\Delta \overrightarrow{\theta}$ werden also iterativ durch folgende Gleichung berechnet und aufsummiert:

$$\Delta \overrightarrow{\theta} = \alpha \mathrm{J}^T \overrightarrow{\mathbf{e}}$$

Der Skalierungsfaktor α wird bei jeder Iteration wie folgt berechnet:

$$\alpha = \frac{\overrightarrow{\mathbf{e}} \cdot \mathrm{J}\mathrm{J}^T \overrightarrow{\mathbf{e}}}{\mathrm{J}\mathrm{J}^T \overrightarrow{\mathbf{e}} \cdot \mathrm{J}\mathrm{J}^T \overrightarrow{\mathbf{e}}}$$

Das Verfahren wird abgebrochen sobald $\overrightarrow{\mathbf{e}}$ klein genug ist oder eine maximale Anzahl an Iterationen überschritten wurde. Eine mögliche Endpose wird in Abbildung 3.8 dargestellt.

3.2.4 Rendering

Nach den in diesem Kapitel beschriebenen Schritten ist in jedem Bone die Matrix „final-Transformation“ gespeichert. Diese Matrizen werden als Uniform-Array an den Vertex-

Shader übergeben, wo die entgültigen Vertex-Positionen berechnet werden.(siehe Abbildung 3.9)

```
void CharacterSkeleton::updateBoneTransforms(Bone3D* bone, const glm::
    ↪ mat4& parentTransform) {
        ...
        Animation& firstAnimation = animations[blendComponents[0].
            ↪ animationName];
        glm::quat rotation = firstAnimation.interpolateRotation(
            ↪ blendComponents[0].animationStatus * firstAnimation.
            ↪ getDuration(), boneName);
        ...
        float weightAcc = blendComponents[0].weight;
        for (unsigned int i = 0; i < blendComponents.size(); ++i) {
                Animation& anim = animations[blendComponents[i].
                    ↪ animationName];
                float cAnimationTime = blendComponents[i].animationStatus *
                    ↪  anim.getDuration();
                if (i != 0) {
                        glm::quat cRotation = anim.interpolateRotation(
                            ↪ cAnimationTime, boneName);
                        rotation = glm::slerp(rotation, cRotation, 1.0f -
                            ↪ weightAcc / (weightAcc + blendComponents[i].
                            ↪ weight));
                        weightAcc += blendComponents[i].weight;
                }
                ..
        }
        bone->rotation = rotation;
        glm::mat4 rotationM = glm::mat4_cast(bone->rotation);
        ...
}
```

Abbildung 3.6: Update der Rotation eines Bones

```
bone->nodeTransformation = translationM * rotationM * scalingM;
...
bone->globalTransformation = parentTransform * bone->nodeTransformation;
bone->finalTransformation = bone->globalTransformation * bone->boneOffset;
    ↪
for (auto& it : bone->children)
       updateBoneTransforms(it, bone->globalTransformation);
```

Abbildung 3.7: Update der Transformations-Matrizen und rekursiver Funktionsaufruf

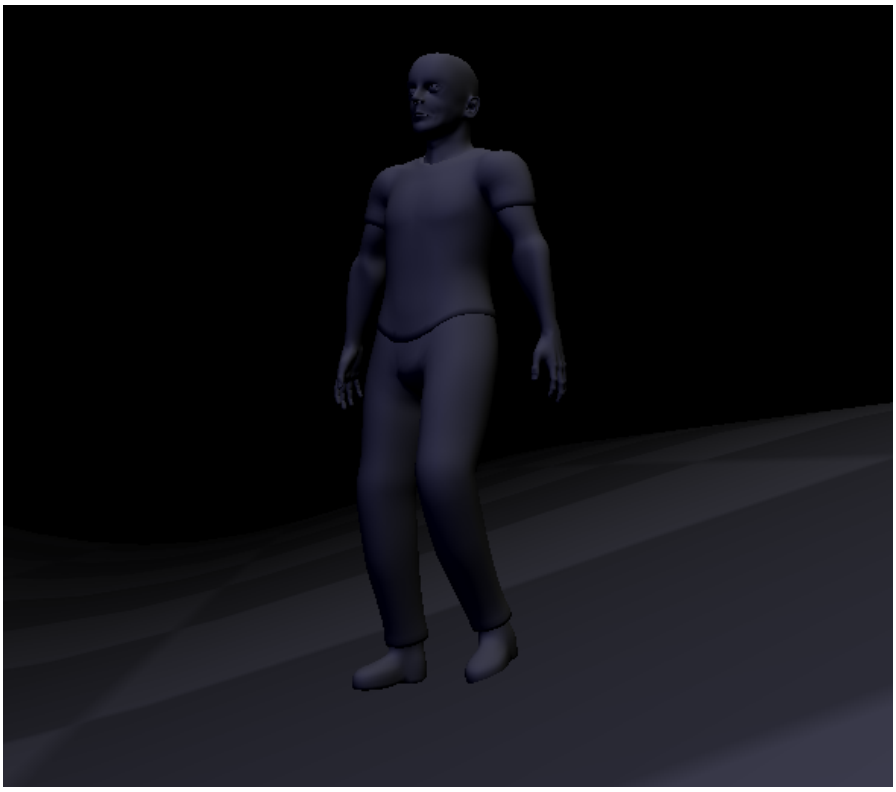


Abbildung 3.8: Pose nach inverse Kinematics

```
...
uniform mat4 bones[100];

void main(void) {
  mat4 boneTransform = bones[in_BoneIds[0]] * in_BoneWeights[0] +
                       bones[in_BoneIds[1]] * in_BoneWeights[1] +
                       bones[in_BoneIds[2]] * in_BoneWeights[2] +
                       bones[in_BoneIds[3]] * in_BoneWeights[3];

  gl_Position = projectionMatrix * modelViewMatrix * boneTransform * vec4(
      ↪ in_Position, 1.0);
  ex_Normal = (modelInverseTranspose * boneTransform * vec4(in_Normal, 0))
      ↪ .xyz;
  ...
}
```

Abbildung 3.9: Vertex-Shader

4 Physik-Simulation

Die im vorherigen Kapitel beschriebenen Verfahren werden nur dann eingesetzt wenn der Charakter sich nicht im Ragdoll-Modus befindet. Ansonsten werden die Rigidbodys von Bullet simuliert und deren Transformations-Matrizen für die Bones übernommen.

4.1 Ragdoll-Modus

Um den Charakter in den Ragdoll-Modus zu wechseln, verfügt die Klasse „CharacterSkeleton“ über die Methode „setRagdollMode“. Sie bewirkt die Aktivierung aller Rigidbodys und Constraints. Im Falle der Rigidbodys muss dafür eine Masse > 0 gesetzt werden.

```
for (auto& it : bonesByName) {
  if (it.second->bHasRigidBody) {
    dynamicsWorld->removeRigidBody(it.second->rigidBody.get());

    btScalar mass = 1;
    btVector3 inertia(0, 0, 0);
    it.second->collisionShape.get()->calculateLocalInertia(mass, inertia);
    it.second->rigidBody->setMassProps(mass, inertia);
    it.second->rigidBody->updateInertiaTensor();
    it.second->rigidBody->setDamping(0.05, 0.85);
    it.second->rigidBody->setDeactivationTime(0.8);
    it.second->rigidBody->setSleepingThresholds(1.6, 2.5);
    it.second->rigidBody->activate();

    dynamicsWorld->addRigidBody(it.second->rigidBody.get());
  }
}
for (auto& it : constraints)
  it.constraint->setEnabled(true);
```

Abbildung 4.1 zeigt das Ergebnis einer Ragdoll-Simulation.

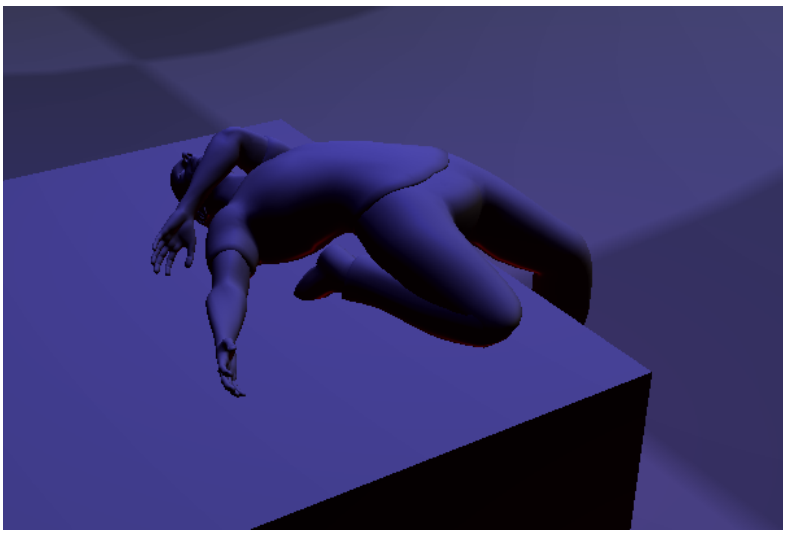


Abbildung 4.1: Pose nach Ragdoll-Simulation

4.2 Cloth-Simulation

Unabhängig vom Ragdoll-Modus wird die Bullet-Engine auch zur Simulation von Cloth-Komponenten des Charakters verwendet. Dazu wird ein „btSoftBody“ mit Hilfe der Funktion„ btSoftBodyHelpers::CreateFromTriMesh“ aus einem Triangle-Mesh erstellt. Die Aufhänge-Vertices werden dabei in Blender durch einen Rotwert von 0 markiert. Diese Vertices erhalten daraufhin eine Masse von 0 und ihre Positionen werden bei jedem Update wie bei normalen Mesh-Komponenten durch forward Kinematics berechnet. Alle anderen Vertices werden von der Bullet-Engine simuliert.

An sich funktioniert dieses Verfahren schon ganz gut, allerdings können unschöne Situationen auftreten wie Abbildung 4.2 zeigt. Um diesen Umstand zu beheben wurde eine Lösung gewählt welche sich an den Motion-Constraints (siehe [Phy]) aus der PhysX-Engine orientiert. Dazu erhält jeder Vertex ein zusätzliches Float-Attribut welches Werte zwischen 0 und 1 annehmen kann. Es beschreibt die maximale Distanz um welche sich der Vertex nach der Physik-Simulation von seiner durch forward Kinematics berechneten Position entfernen darf. Für dieses Attribut wird der Rot-Wert des Vertex verwendet. In Abbildung 4.3 werden diese Distanzen visualisiert. Der zuvor gezeigte Fall kann nun nicht mehr auftreten.

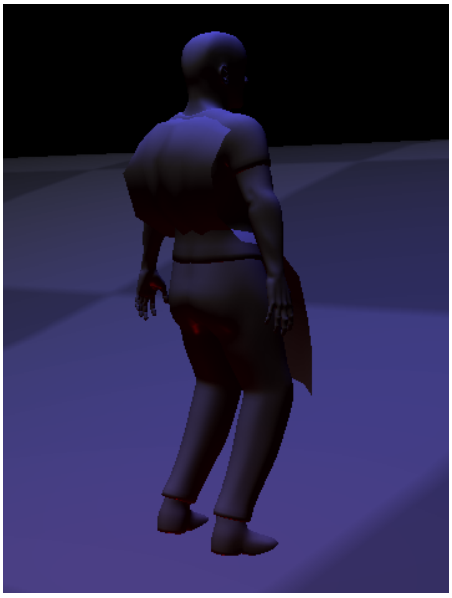


Abbildung 4.2: Situation nach schneller Drehung

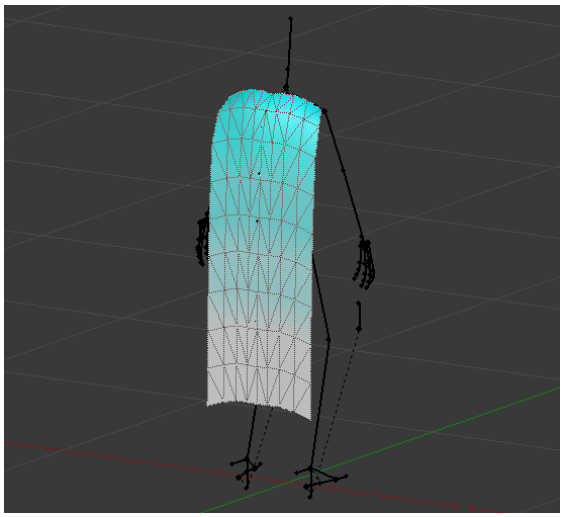


Abbildung 4.3: Vertex-Farbe für maximale Distanzen

5 Zusammenfassung

Die meisten der vorgenommenen Ziele konnten umgesetzt werden. Die Bullet-Engine scheint für diese Zwecke also bestens geeignet zu sein. Die Softbody-Simulation könnte allerdings etwas robuster sein. Ohne die Distanz-Constraints kam es gelegentlich zu Explosionen des Softbodys. Außerdem ist ein leichtes Zittern der Vertices feststellbar wenn der Softbody auf dem Boden aufliegt. Die Ragdoll-Animation sieht ebenfalls nicht ganz natürlich aus wenn Softbodys beteiligt sind. Allerdings könnte dies möglicherweise durch Anpassung der Parameter verbessert werden.

Denkbare zukünftige Verbesserungen wären:

- Die Verwendung der neuen GPU Rigidbody Pipeline von Bullet 3.
- Einsatz einer besseren Methode für inverse Kinematics, z.B. selectively damped least squares nach [Bus04].
- Die Implementation von Dual-Quaternion-Blend-Skinning nach [Kav07] statt linearem Blend-Skinning.

Der gesamte Quelltext ist auf Github unter folgender Url verfügbar:
https://github.com/Maxjen/Calamity

Literatur

[Bus04] S. R. Buss. *Selectively Damped Least Squares for Inverse Kinematics*. Techn. Ber. University of California, 2004.

[Bus09] S. R. Buss. *Introduction to Inverse Kinematics with Jacobian Transpose, Pseudoinverse and Damped Least Squares methods*. Techn. Ber. University of California, 2009.

[Gue] A. Guenette. *Blenderella, Character Modeling in Blender 2.5*. URL: http://www.blender3d.org/e-shop/product_info_n.php?products_id=133.

[Kav07] L. Kavan. *Skinning with Dual Quaternions*. Techn. Ber. Trinity College Dublin, Czech Technical University in Prague, 2007.

[Mei] E. Meiri. *OGLdev Modern OpenGL Tutorials*. URL: http://ogldev.atspace.co.uk/.

[Phy] PhysX-Entwickler. *PhysX Manual*. URL: https://developer.nvidia.com/sites/default/files/akamai/physx/Manual/Cloth.html#motion-constraints.