2014

Improving the performance of skeletal mesh animations in the Blender game engine

Mitchell Stokes
Eastern Washington University

Improving the Performance of Skeletal Mesh Animations in the Blender Game Engine

---

A Thesis

Presented To

Eastern Washington University

Cheney, Washington

In Partial Fulfillment of the Requirements

for the Degree

Master of Science in Computer Science

By

Mitchell Stokes

Winter 2014

Thesis of Mitchell Stokes Approved by

DR. PAUL SCHIMPF, GRADUATE STUDY COMMITTEE	DATE
STUART STEINER, GRADUATE STUDY COMMITTEE	DATE

Master's Thesis

In presenting this thesis in partial fulfillment of the requirements for a masters degree at Eastern Washington University, I agree that the JFK Library shall make copies freely available for inspection. I further agree that copying of this project in whole or in part is allowable only for scholarly purposes. It is understood, however, that any copying or publication of this thesis for commercial purposes, or for financial gain, shall not be allowed without my written permission.

SIGNATURE DATE

Contents

List of Figures

1 Introduction

Blender is a free and open-source 3D modeling and animation program [1]. One of its many features is a built-in game engine called The Blender Game Engine (BGE). Character animations in the BGE are considered slow by the BGE community. For comparison, the benchmark scene described in section 3.3 runs at approximately 187ms per frame on the benchmark machine described in section 3.2. As a quick test, a scene similar to the benchmark scene was created and ran using and open-source graphics engine called OGRE [2]. In OGRE, the scene ran at approximately 37ms per frame. Not only is this approximately five times faster, but the OGRE test could probably be further optimized by someone more knowledgeable with OGRE. The OGRE example uses just the basics needed to get the scene to run. For example, different scenegraphs, instancing or hardware skinning could be explored and used.

This work attempts to improve the performance of these skeletal mesh animations in the BGE. Several optimizations are used in this work, which can be grouped into three main categories: parallelizing code, moving code to the GPU, and other minor optimizations.

2 Background

2.1 OpenGL and the GPU

Games (and most any real-time applications) leverage the impressive power of modern graphics processing units (GPUs). In the past, GPUs were almost entirely used for graphics processing as the name would suggest, however general purpose GPU computing (GPGPU) has also become popular. OpenGL is an application programming interface (API) that allows developers to communicate with graphics hardware.

As graphics hardware has progressed, OpenGL has also needed to expand, and thus has different versions. As of the time of this writing, OpenGL is up to version 4.4. However, one must keep in mind that OpenGL versions only specify a minimum specification that graphics hardware must meet. For example, an OpenGL 2 capable graphics card may still be able to use some OpenGL 3 features. Therefore, when using OpenGL, it is common to target features instead of OpenGL versions (e.g., check for shader support instead of checking for OpenGL 2 support).

In earlier versions of OpenGL, a fixed graphics pipeline was defined. This pipeline could be setup, used and influenced by commands, but certain parts (e.g., the algorithm to determine pixel color) were still fixed. Eventually, parts of the pipeline became programmable. The programs written to control these parts of the pipeline are called shaders. OpenGL 4.4 specifies four shader types: vertex, geometry, fragment, and compute. Primarily GLSL is used to write these shaders, but other languages such as Cg (a high-level shading language from NVIDIA) are supported. Of interest to this work are the vertex and (partially) the fragment shader. A vertex shader controls vertex transformations (e.g., the model to view space coordinate conversion), and a fragment shader controls how a fragment (think pixel) is colored or shaded.

2.1.1 GPU/CPU Interaction

When using a GPU, the central processing unite (CPU) is still used. This means that two processors are being used asynchronously and in parallel. The CPU is used to issue commands to the GPU (usually in some form of render step) Usually the CPU can continue to work while the GPU processes the commands it was given. However, the GPU driver can also block the CPU if, for example, the GPU has too much work to do. This gives two general categories of bottlenecks in a graphics application:

1. The CPU is not issuing commands to the GPU fast enough. This results in the GPU being underutilized, and is known as being CPU-bound.

2. The CPU is too far ahead of the GPU and the GPU drivers cause the CPU to wait. In this case the CPU is being underutilized, and this is known as being GPU-bound.

Improving the performance of code on running on the CPU will not improve performance if the CPU is just going to wait longer for the GPU to finish (GPU-bound). And similarly, if the application is CPU-bound, giving the GPU less work will also not improve performance. Therefore, it is important to identify the bottleneck and keep the load on the CPU and GPU balanced for the best performance.

2.2 Skeletal Mesh Animation

One popular method used to drive character animations is to use a set of bones (also known as joints) called a skeleton (or armature) to deform a mesh. An example of this can be seen in figure 1.

Figure 1: An example skeletal mesh

Once a pose (a set of transforms for each bone) has been determined, an operation called skinning is used to deform the mesh to the pose. Transformations are usually represented as a four-by-four matrix, which encodes rotation, translation and scale. The pose transformations are relative to an initial "rest" pose. Each vertex in the mesh has a set of weights, which sum to one, to determine the amount of influence each bone has on that vertex. To determine each vertex's new skinned position and normal, equations 1 and 2 can be used.

$$P' = \sum_b (w_b T_b P) \tag{1}$$

$$N' = \sum_b (w_b T_{rb} N) \tag{2}$$

For the position equation (equation 1) P is the initial, rest position of the vertex, b is a bone, w_b is the scalar influence (i.e., the weight) of b on the vertex and T_b is the four-by-four transform matrix of the bone. The normal equation (equation 2) is similar except we use the normal's resting rotation (N) and the three-by-three rotation part of the bone transformation matrix as T_{rb}. These calculations must be done every time the pose of the skeleton is updated to create a new set of positions and normals for a mesh's vertexes. When rendering the mesh this also requires the vertex data to be re-sent to the graphics card if the skinning calculations were done on the CPU (software skinning).

Figure 2 shows an example skeletal mesh animation in Blender, which uses a right-handed, Z-up coordinate system. This means that the up axis is the positive Z axis, the right is the positive X axis, and going into the screen is the positive Y axis. The example is a rectangular prism centered on the origin.

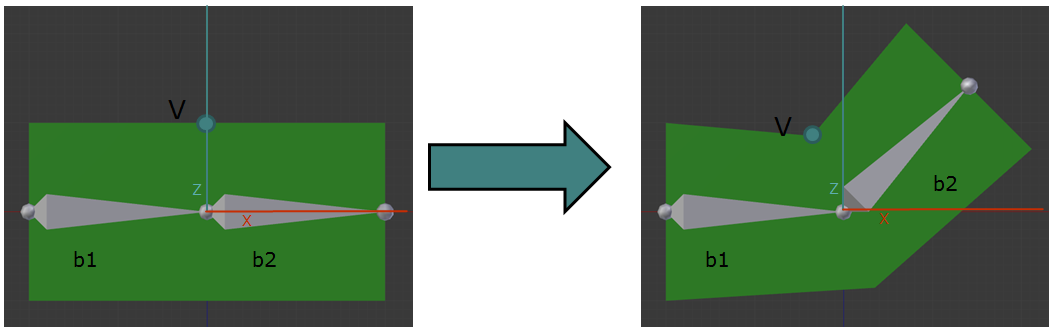

Figure 2: An example skeletal mesh animation

Using equation 1, a skinned position can be determined for the vertex labeled V due to the animation shown:

$$V = \begin{pmatrix} 0.0 \\ -1.0 \\ 1.0 \\ 1.0 \end{pmatrix} T_{b1} = \begin{pmatrix} 1.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 1.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 1.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 1.0 \end{pmatrix} T_{b2} = \begin{pmatrix} 0.7071 & 0.0 & -0.7071 & 0.0 \\ 0.0 & 1.0 & 0.0 & 0.0 \\ 0.7071 & 0.0 & 0.7071 & 0.0 \\ 0.0 & 0.0 & 0.0 & 1.0 \end{pmatrix}$$

$$w_{b1} = 0.5$$

$$w_{b2} = 0.5$$

$$V' = \sum_b (w_b T_b V)$$

$$V' = w_{b1} T_{b1} V + w_{b2} T_{b2} V$$

$$V' = 0.5 \begin{pmatrix} 1.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 1.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 1.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 1.0 \end{pmatrix} \begin{pmatrix} 0.0 \\ -1.0 \\ 1.0 \\ 1.0 \end{pmatrix} + 0.5 \begin{pmatrix} 0.7071 & 0.0 & -0.7071 & 0.0 \\ 0.0 & 1.0 & 0.0 & 0.0 \\ 0.7071 & 0.0 & 0.7071 & 0.0 \\ 0.0 & 0.0 & 0.0 & 1.0 \end{pmatrix} \begin{pmatrix} 0.0 \\ -1.0 \\ 1.0 \\ 1.0 \end{pmatrix}$$

$$V' = \begin{pmatrix} -0.35355 \\ -1.0 \\ 0.85255 \\ 1.0 \end{pmatrix} \begin{matrix} (left) \\ (forward) \\ (up) \end{matrix} \tag{3}$$

A technique referred to as hardware skinning moves the skinning step from the CPU to the GPU. This also allows the skinning calculation to benefit from the impressive parallelization offered by modern graphics hardware. Another benefit is that the skeletal mesh can be treated as static geometry since all of the deformations are done on the GPU. This allows options such as display lists and vertex buffer objects [3] to be used to increase rendering performance.

The differences in program flow between software and hardware skinning are shown in figure 3.

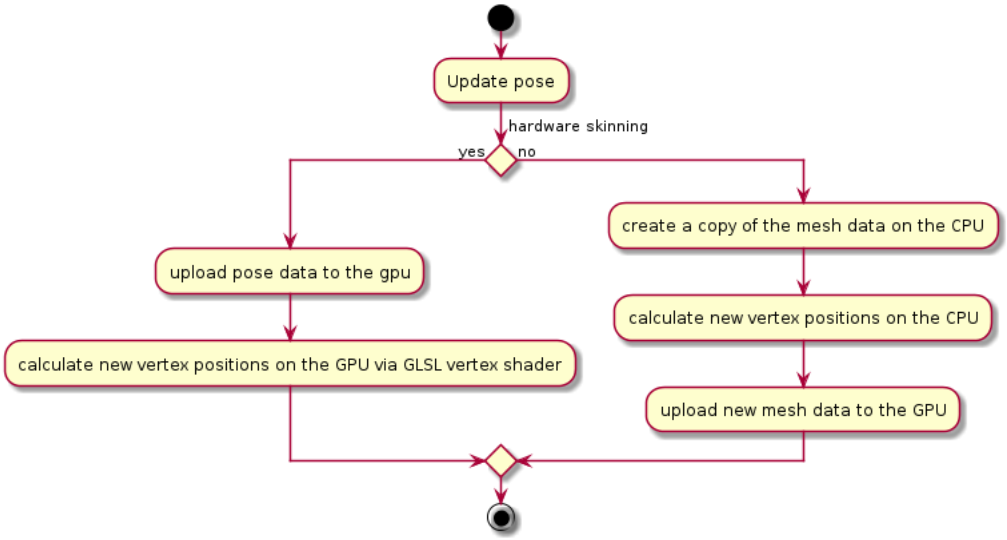

Figure 3: UML Activity diagram for skinned animations

As can be seen in figure 3, instead of sending new vertex data every frame, the pose data must be sent every frame. This means, as far as memory bandwidth, $number\_of\_verts * sizeof(vertex)$ is being traded for $number\_of\_bones * sizeof(bone_transform)$. The size of a vertex in the BGE can range from between six floats (position and normal) to thirty-one floats (position, normal, color, and various texture coordinates). Using just position, normals and UV texture coordinates results in only eight floats and is feasible for a simple character. Bone transformations, as mentioned earlier, are a four-by-four matrix, which is sixteen floats. As for the number of vertexes and bones, documentation for Unity3D (a game engine) notes that skeletons will usually have between sixteen and sixty bones, and meshes will usually have between 1,500 to 4,000 vertexes, but possibly upwards of 5,000 to 7,000 vertexes [4]. Using thirty bones as suggested in [4] and an average of 2,750 vertexes, the total memory, M, needed for vertex data is (assuming four bytes per float):

$$
\begin{aligned}
M &= number\_of\_verts * sizeof(vertex) \\
M &= 2,750 * 8 * sizeof(float) \\
M &= 88,000 \; bytes
\end{aligned}
$$

and for pose data is:

$$
\begin{aligned}
M &= number\_of\_bones * sizeof(bone\_transform) \\
M &= 30 * 16 * sizeof(float) \\
M &= 1,920 \; bytes
\end{aligned}
$$

In other words, hardware skinning will typically not use more memory bandwidth than software skinning.

Hardware skinning is usually implemented as a vertex shader. An example GLSL vertex shader from version 9.52 of NVIDIA's graphics SDK [5] is shown in listing 1.

Listing 1: Example vertex shader for hardware skinning from NVIDIA's graphics SDK

```
attribute vec4 position;
attribute vec3 normal;
attribute vec4 weight;
attribute vec4 index;
attribute float numBones;
```

```
uniform mat4 boneMatrices[30];
uniform vec4 color;
uniform vec4 lightPos;

void main()
{
    vec4 transformedPosition = vec4(0.0);
    vec3 transformedNormal = vec3(0.0);

    vec4 curIndex = index;
    vec4 curWeight = weight;

    for (int i = 0; i < int(numBones); i++)
    {
        mat4 m44 = boneMatrices[int(curIndex.x)];

        // transform the offset by bone i
        transformedPosition += m44 * position * curWeight.x;

        mat3 m33 = mat3(m44[0].xyz,
                        m44[1].xyz,
                        m44[2].xyz);

        // transform normal by bone i
        transformedNormal += m33 * normal * curWeight.x;

        // shift over the index/weight variables, this moves the index and
        // weight for the current bone into the .x component of the index
        // and weight variables
        curIndex = curIndex.yzwx;
        curWeight = curWeight.yzwx;
    }

    gl_Position = gl_ModelViewProjectionMatrix * transformedPosition;

    transformedNormal = normalize(transformedNormal);
    gl_FrontColor = dot(transformedNormal, lightPos.xyz) * color;
}
```

Attributes are per-vertex values while uniforms are global values. Typically uniforms will be changed per-material or per-mesh. The NVIDIA example has a maximum of four bones influencing each vertex, which allows weights and indexes (into the boneMatrices array) to be stored as vec4 (four floats) data types. It also imposes a limit of thirty bones for the whole skeleton. In the NVIDIA shader, the mat44 variable stores a bone's four-by-four transform matrix, which is multiplied against the current vertex's vec4 position value (the part inside the summation of equation 1). The mat33 variable is a three-by-three orientation matrix constructed from the bone's transform matrix, and it is multiplied against the current vertex's vec3 normal value (the part inside the summation of equation 2). A for loop is used for the summation, and the resulting vectors are added back to the initial position and normal vectors.

2.3 Blender and the BGE

2.3.1 Class Prefixes

Blender and the BGE are broken down into many modules. Functions and classes from these modules often have a prefix to help identify them (these prefixes also act as a poor-man's namespace to avoid name conflicts). In the BGE the following prefixes can be found:

BL Items closely tied to Blender. Most of these items are part of the code that converts Blender data to BGE data.

SCA Generic game engine code. This handles a lot of generic logic and most of the BGE's logic brick system (a visual programming system). The prefix comes from the BGE's Sensor, Controller, Actuator architecture for logic bricks.

KX These items tend to be BGE specific code. The name comes from Ketsji, which was an old name for the BGE.

KX_SCA These items blur the line between SCA and KX items. They should probably belong in one prefix or the other.

SG These items belong to the BGE's scenegraph code.

RAS These items belong to the BGE's rasterization, or rendering, code. This code is usually referred to as the "rasterizer."

PHY These items belong to the BGE's generic physics interface.

Some prefixes from Blender that one might see used in BGE code include:

BLF Blender font handling code.

BKE Blender kernel, code. This contains most of the code to actually manipulate Blender data.

BLI Contains useful data structures such as linked lists, trees, etc.

BLO Blender file loading code.

GPU Blender GPU rendering code that is primarily used for the viewport. This module is sometimes referred to as bf_gpu when talking about rendering to differentiate it from general gpu code/programming (e.g., OpenGL).

DNA Blender struct definition. Refer to section 2.3.2 for more details.

RNA Description layer for DNA. Refer to section 2.3.2 for more details.

2.3.2 Blender and DNA/RNA

Blender contains two systems for storing and manipulating data, these are DNA and RNA. DNA is responsible for storing (and ultimately saving) Blender data, while RNA describes the DNA data and how it can be accessed. In other words, DNA values (stored as C structs) are modified via RNA (functions). Blender's animation system uses this RNA for writing out new pose data. Unfortunately looking up RNA values and modifying them were never optimized for speed, and sometimes these operations can be slow.

2.3.3 BGE Overview

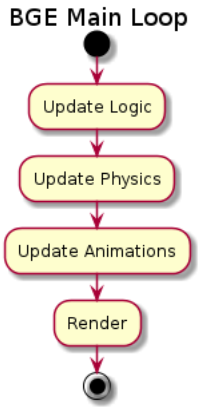

Figure 4: UML activity diagram for the BGE's main loop

Figure 4 shows a simplified version of the BGE's main loop. As can be seen, every frame the BGE has roughly four main tasks: update logic, update physics, update animations, and render. When updating logic, the engine evaluates and executes any logic bricks or Python scripts that a user may have setup. The physics engine (the BGE uses Bullet) is then given the chance to evaluate the physics scene and step its simulations. Animations are updated next, which is further explained in section 2.3.4. Finally, the scene must be rendered. This involves determining what needs to be rendered (objects that will not be rendered this frame are "culled"), and issuing the appropriate commands to have the GPU render the scene. Excluding the actual rasterization of the scene, everything is performed on the CPU (including culling and mesh deformations), and in a single thread.

2.3.4 BGE Animation Code

The code for updating animations can be seen in listing 2.

Listing 2: BGE animation code

```
void KX_Scene::UpdateAnimations(double curtime)
{
    // Update any animations
    for (int i=0; i<m_animatedlist->GetCount(); ++i)
        ((KX_GameObject*)m_animatedlist->GetValue(i))->UpdateActionManager(curtime);
}
```

To handle animations, the BGE makes use of various classes, which are shown in figure 5. For discussion purposes, the prefixes will be left off of the class names.

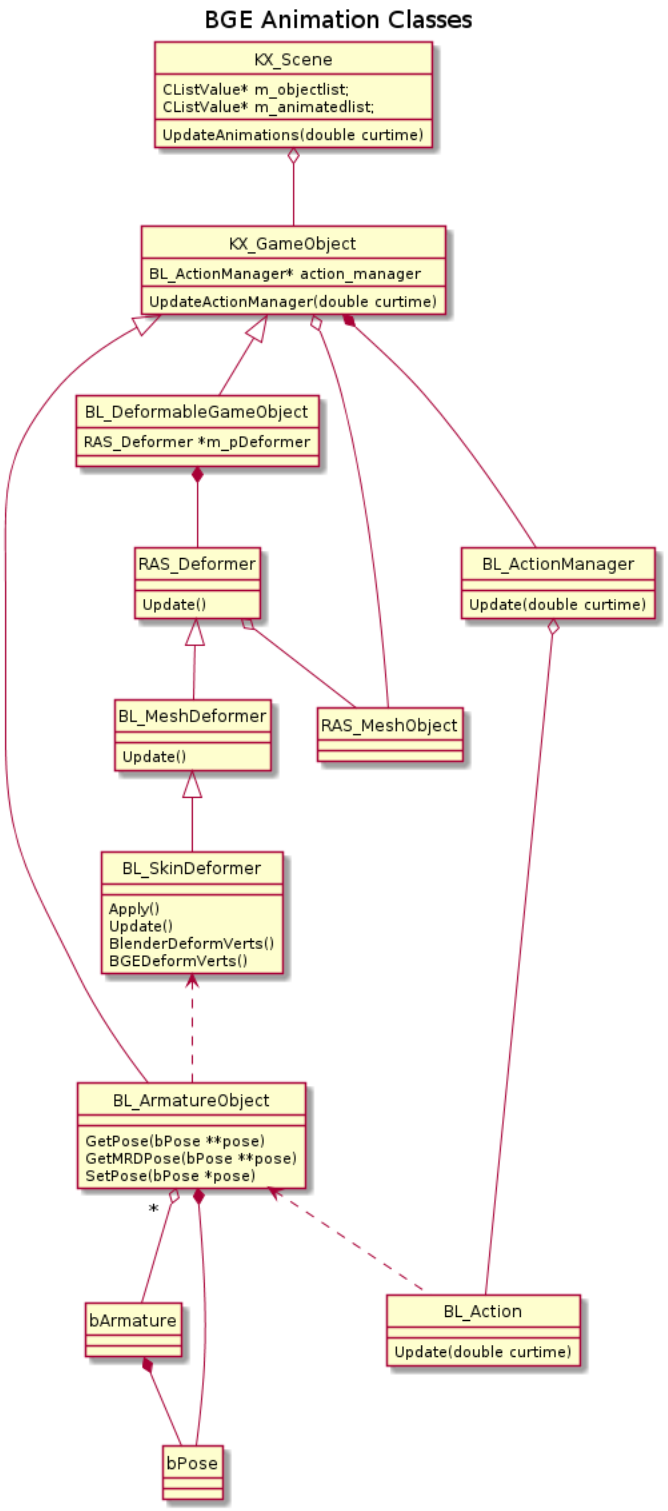

Figure 5: UML class diagram of classes needed for animations in the BGE

The Scene class contains an aggregate, called m_objectlist, of all of the GameObjects in the scene (CListValue is a wrapper around C++'s vector class). Some of those objects can also be present in m_animatedlist, which is iterated in Scene::UpdateAnimations(). ArmatureObjects represent skeletons that we may or may not want to animate, and DeformableObjects are ones with deform-able

meshes. A handful of Deformers exist in the BGE, but the SkinDeformer is the only one of interest for skeletal mesh animations. The SkinDeformer will deform the mesh based off of the current pose of an ArmatureObject (usually the mesh's parent object). A Deformer is updated during the render stage while ArmatureObjects are updated during the animation stage.

The ArmatureObject contains Blender data in the form of bArmature and bPose pointers. Every ArmatureObject that uses the same skeleton will also point to the same bArmature. All ArmatureObjects make copy of their bArmature's bPose. In order to actually update the pose, the bArmature's bPose pointer is saved and replaced with the ArmatureObject's bPose, and then some Blender functions are called to animate the bArmature. The bArmature's original pointer is then restored. The code that performs this update (as well as some blending) is shown in listing 3.

Listing 3: Pose update performed as part of BL_Action::Update()

```
if (m_obj->GetGameObjectType() == SCA_IObject::OBJ_ARMATURE)
{
    BL_ArmatureObject *obj = (BL_ArmatureObject*)m_obj;
    obj->GetPose(&m_pose);

    // Extract the pose from the action
    {
        Object *arm = obj->GetArmatureObject();
        bPose *temp = arm->pose;

        arm->pose = m_pose;

        PointerRNA ptrrna;
        RNA_id_pointer_create(&arm->id, &ptrrna);

        animsys_evaluate_action(&ptrrna, m_action, NULL, m_localtime);

        arm->pose = temp;
    }

    // Handle blending between armature actions
    if (m_blendin && m_blendframe<m_blendin)
    {
        IncrementBlending(curtime);

        // Calculate weight
        float weight = 1.f - (m_blendframe/m_blendin);

        // Blend the poses
        game_blend_poses(m_pose, m_blendinpose, weight, ACT_BLEND_BLEND);
    }


    // Handle layer blending
    if (m_layer_weight >= 0)
    {
        obj->GetMRDPose(&m_blendpose);
        game_blend_poses(m_pose, m_blendpose, m_layer_weight, m_blendmode);
    }

    obj->SetPose(m_pose);

    obj->SetActiveAction(NULL, 0, curtime);
}
```

The SkinDeformer contains two functions for handling the mesh deformations that are exposed to the user as "Vertex Deformers": BlenderDeformVerts() and BGEDeformVerts(). Originally the SkinDeformer would make calls into Blender code to handle the deformation. Later a "BGE Vertex Deformer" (the original code was used to create the BlenderDeformVerts() function) was added to focus on speed over accuracy, and it decreases the frame time of an animation heavy scene by about

30%. However, it lacks some features such as support for Blender's B-Bones, and it has less accurate normal calculations. These vertex deformers are called as part of SkinDeformer::Update(), which, in turn, is called as part of SkinDeformer::Apply(). This makes SkinDeformer::Apply() the function to call to kickoff skinning in the BGE.

When setting up a skeletal mesh in Blender, a mesh and skeleton are first both created. Then, the mesh is "parented" to the skeleton. Thus, a mesh has at most one skeleton deforming it, but a skeleton can deform many meshes. However, in most cases this is simply a one-to-one relationship.

Other types of animations other than skeletal mesh animations can be performed. These include changing an object's color, changing an object's transform (position, rotation and/or scale), and other minor animations. Since these animations are simple (usually updating a single vector or matrix), they are usually much faster than animating a skeletal mesh. As such, this work will only focus on improving the performance of skeletal mesh animations, and ignore the performance of other types of animations.

3 Methods

3.1 Version Control

Blender recently switched from Subversion (SVN) [6] to Git [7] as its version control system [8]. The main Git branch for Blender is its "master" branch. This project makes use of a repository cloned to GitHub [9], where a "thesis" branch was also created.

3.2 Benchmark Platform

The following table lists various details of the computer used to run the benchmark scene.

Software	
Operating System	Arch Linux 64bit 3.13.4 kernel
Compiler	GCC 4.8.2
Graphics Driver	NVIDIA Proprietary 331.38
Hardware	
CPU	Intel Core i7 Q 740 @ 1.73GHz
RAM	8GB of DDR3
GPU	NVIDIA GeForce GT 425M

3.3 Benchmark Scene

The scene used to benchmark performance changes is comprised of 160 skeletal meshes. Each of these skeletal meshes contains a sixty-nine bone armature and a mesh with 4,951 vertexes. All 160 skeletal meshes perform the same animation, which is a silly dance animation. However they each have their own skeleton, and each skeleton's pose update is done independent of other skeletons. Therefore, this benchmark can still be representative of multiple different characters performing different actions. The scene also has no physics being calculated, which allows the characters to freely clip into one another. A screenshot of the scene being run is shown in figure 6.

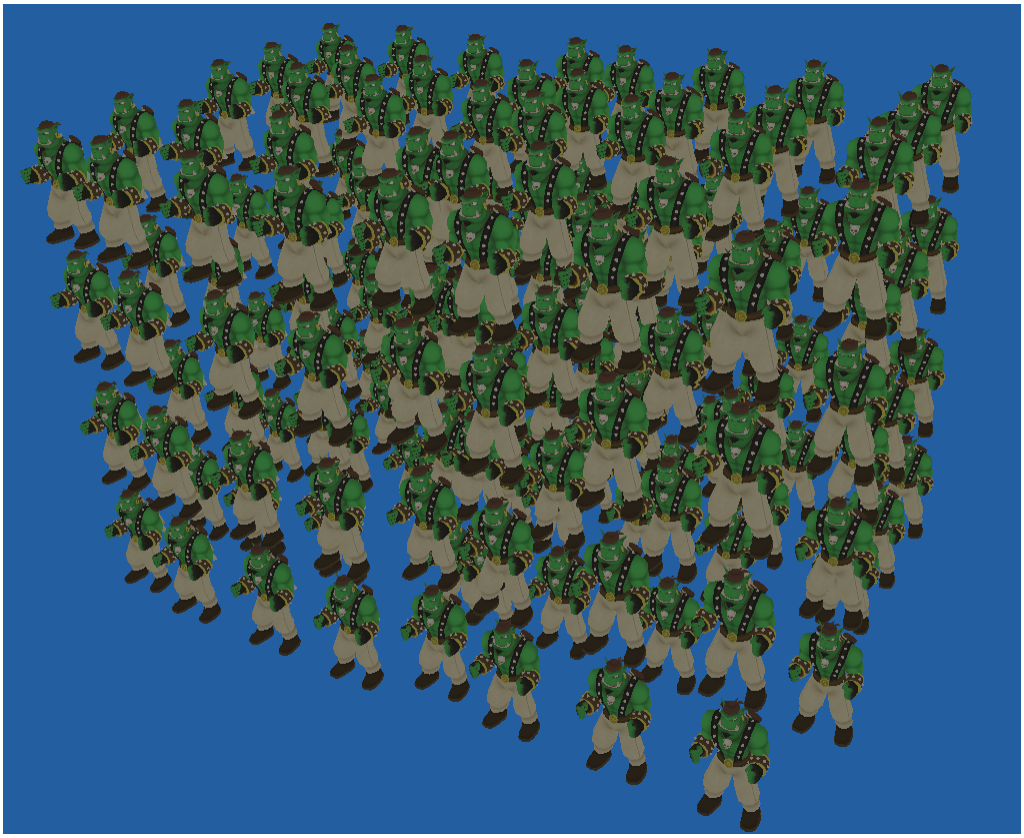

Figure 6: Screenshot of the benchmark scene

The benchmark scene will be run through the stand-alone Blenderplayer instead of the embedded player using the following settings:

```
blenderplayer -w 1600 900 -g show_framerate = 1 -g show_profile = 1
anim_stress.blend
```

Another version of this scene, which is accessed by adding `- random` to the end of the above command line string, assigns a random action to each character. While this version is not as consistent as the original scene (making it poorer for benchmarking), it does help to visually verify that duplicates of the same character can play different actions. In other words, to show that the threaded code is not breaking when used with different actions.

3.4 Measuring Performance

3.4.1 BGE Profiler

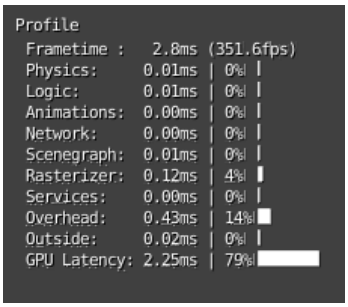

Figure 7: Screenshot of the BGE's in-game profiler

Figure 7 shows a screenshot of the BGE's in-game profiler. The frametime is split into the following categories: physics, logic, animations, network, scenegraph, rasterizer, services, overhead, outside, and GPU latency. Of importance to this work are: animations, rasterizer, and GPU latency. In general, the animation category logs time spent on pose and – after the work in section 4.3.2 – software skinning updates. The rasterizer category logs time spent on getting data ready to send to the GPU. This includes actions such as culling, sorting meshes for alpha, and sending buffer data to the GPU. GPU latency is the time spent on a SwapBuffers calls. This gives an idea of how long the CPU has to spend waiting for the GPU to finish rendering. [10] gives a more detailed overview of the various categories.

For this work, the profiler is mostly used to get frametime and fps numbers.

3.4.2 gperftools

gperftools [11] is used for more detailed profiling that is offered by the BGE's profiler. The data from gperftools can also be used to generate a visual callgraph.

3.4.3 nvidia-settings

The nvidia-settings utility that ships with the NVIDIA drivers can be used to configure the driver and query the driver for information. Starting with version 331.20 of the driver, nvidia-settings can query for GPU utilization [12]. The following command is used to monitor GPU utilization:

```
watch -tn1 nvidia-settings -q gpuutilization
```

3.5 Verification

Helgrind (part of the Valgrind suite [13]) can be used to check for threading errors. According to the Helgrind manual [14], Helgrind can detect the following three classes of errors:

1. Misuses of the POSIX pthreads API
2. Potential deadlocks arising from lock ordering problems
3. Data races – accessing memory without adequate locking or synchronization

Since OpenMP is being used for parts of this work, as noted in section 7.5 "Hints and Tips for Effective Use of Helgrind" of the Helgrind manual [14], GCC needs to be built with the `--disable-linux-futex` option for accurate output when using OpenMP.

Another Valgrind tool, Memcheck [15], was used to check for memory errors.

4 Work

4.1 Animation Culling

The original animation code shown in listing 2 does not take into account whether the animated object will actually be visible. This means that animation updates can be performed that have no visual impact, but still consume resources. Regular object animations are much quicker in comparison to skeletal mesh animations, which means culling them is not as important. This simplifies the culling calculation since we can make some assumptions about skeletal meshes. A skeletal mesh should have an armature and zero or more children meshes. If the armature has any children meshes and all of these child meshes were culled, then we do not need to bother with updating the pose or the mesh since they will not be visible. An example of what this code (without any of the parallel improvements) looks like is provided in listing 4.

Listing 4: BGE animation code with culling

```
void KX_Scene::UpdateAnimations(double curtime)
{
    for (int i=0; i<m_animatedlist->GetCount(); ++i) {
        KX_GameObject *gameobj, *child;
        bool needs_update;

        gameobj = (KX_GameObject*)m_animatedlist->GetValue(i);

        // Non-armature updates are fast enough, so just update them
        needs_update = gameobj->GetGameObjectType() != SCA_IObject::OBJ_ARMATURE;

        if (!needs_update) {
            // If we got here, we're looking to update an armature, so check its
            // children meshes to see if we need to bother with a more expensive
            // pose update
            CListValue *children = gameobj->GetChildren();

            bool has_mesh = false, has_non_mesh = false;

            // Check for meshes that haven't been culled
            for (int j=0; j<children->GetCount(); ++j) {
                child = (KX_GameObject*)children->GetValue(j);

                if (!child->GetCulled()) {
                    needs_update = true;
                    break;
                }

                if (child->GetMeshCount() == 0)
                    has_non_mesh = true;
                else
                    has_mesh = true;
            }

            // If we didn't find a non-culled mesh, check to see
            // if we even have any meshes, and update if this
            // armature has only non-mesh children.
            if (!needs_update && !has_mesh && has_non_mesh)
                needs_update = true;

            children->Release();
        }

        if (needs_update)
            gameobj->UpdateActionManager(curtime);
    }
}
```

This extra culling check adds very little to the overall cost of the animation code since mesh culling is already being done. However, if some pose and mesh updates can be avoided a fair amount of time can be saved. The exact savings is difficult to calculate since it is view dependent.

This optimization has already been committed to Blender's master branch and was included as part of the 2.69 release.

4.2 Bundled RNA Lookups

As discussed in section 2.3.2, updating poses requires using RNA to modify values, which can be slow. Looking up the RNA property is one of the more expensive parts of the RNA writing step. Every channel (e.g., X Position, Y Position, Z Position, etc) was being looked up as a separate property. However, RNA supports looking up and modifying an array (e.g., Position). Therefore, time can be saved by modifying arrays (one lookup per three to four writes) at a time instead of each channel individually (one lookup per one write). The performance improvement is shown in figure 8.

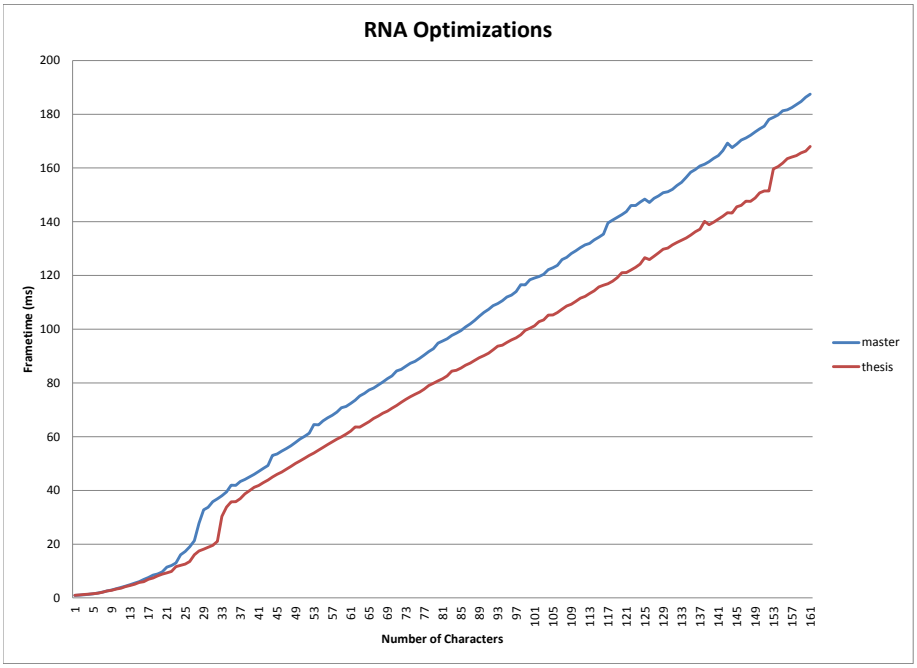

Figure 8: Results of RNA improvements

4.3 Multithreading

4.3.1 Parallel Pose Updates

The juggling of the bPose reference of bArmature described in section 2.3.4 saves memory, but it can cause race conditions when attempting to perform the pose updates in parallel. If multiple ArmatureObjects share a single reference to a bArmature, and then those ArmatureObjects attempt to modify their bArmature simultaneously, problems will occur. Therefore, in order to avoid the race conditions, it is better that each ArmatureObject not only has a unique copy of the bPose, but a unique copy of the bArmature as well. The copy does consume a little more memory, but it makes threading the animation code possible. Another potential race condition involved the bAction data stored by the Action class. This bAction data does get modified during animation updates, so it was best that each Action object held its own copy of the bAction data to avoid potential race conditions.

Initially, OpenMP [16] was used to parallelize this loop using an OpenMP parallel for pragma. While OpenMP was simple to use, OpenMP support is compiler dependent since it relies on pragmas. Since OpenMP is implemented at the compiler level, support can vary (e.g., different OpenMP

versions), and performance is not equivalent across compilers. For example, MSVC 2010 introduced a performance regression with OpenMP code that could cause code to run two times slower than MSVC 2008 [17].

Recently, a threaded task scheduler was added to Blender named BLI_task [18]. This opened up an alternative to OpenMP that was not as compiler dependent. Furthermore, BLI_task could be used for other parts of the BGE. Therefore, the code was switched from OpenMP to BLI_task, which can be seen in listing 5. There was no discernible difference in performance between BLI_task and OpenMP on the test platform (section 3.2).

Listing 5: Multithreaded animation update code

```
static void update_anim_thread_func(TaskPool *pool,
                                    void *taskdata,
                                    int UNUSED(threadid))
{
    KX_GameObject *gameobj, *child;
    CListValue *children;
    bool needs_update;
    double curtime = *(double*)BLI_task_pool_userdata(pool);

    gameobj = (KX_GameObject*)taskdata;

    // Non-armature updates are fast enough, so just update them
    needs_update = gameobj->GetGameObjectType() != SCA_IObject::OBJ_ARMATURE;

    if (!needs_update) {
        // If we got here, we're looking to update an armature, so check its
        // children meshes to see if we need to bother with a more expensive
        // pose update
        children = gameobj->GetChildren();

        bool has_mesh = false, has_non_mesh = false;

        // Check for meshes that haven't been culled
        for (int j=0; j<children->GetCount(); ++j) {
            child = (KX_GameObject*)children->GetValue(j);

            if (!child->GetCulled()) {
                needs_update = true;
                break;
            }

            if (child->GetMeshCount() == 0)
                has_non_mesh = true;
            else
                has_mesh = true;
        }

        // If we didn't find a non-culled mesh, check to see
        // if we even have any meshes, and update if this
        // armature has only non-mesh children.
        if (!needs_update && !has_mesh && has_non_mesh)
            needs_update = true;

        children->Release();
    }

    if (needs_update)
        gameobj->UpdateActionManager(curtime);
}

void KX_Scene::UpdateAnimations(double curtime)
{
    TaskPool *pool = BLI_task_pool_create(KX_GetActiveEngine()->GetTaskScheduler(),
                                          &curtime);
```

```
    for (int i=0; i<m_animatedlist->GetCount(); ++i) {
        BLI_task_pool_push(pool,
                           update_anim_thread_func,
                           m_animatedlist->GetValue(i),
                           false,
                           TASK_PRIORITY_LOW);
    }

    BLI_task_pool_work_and_wait(pool);
    BLI_task_pool_free(pool);
}
```

4.3.2 Parallel Software Skinning

The skinning step was not already in an easy-to-parallelize loop like the pose update. Another problem with the software skinning is that it is counted under the "Rasterizer" category in the profiler discussed in section 3.4.1. It would be ideal to perform the skinning step in the same loop as the pose updates (Scene::UpdateAnimations()), as this solves both problems (the loop is already running in parallel, and is already being recorded as "Animation" time). As discussed in section 2.3.4, SkinDeformer::Apply() is essentially the entry point for the software skinning.

Unfortunately, SkinDeformer::Apply() requires a IPolyMaterial pointer, which is part of the rasterizer code and not something that Scene::UpdateAnimations() has access to. However, SkinDeformer::Update() has no arguments, and none of the skinning code requires the IPolyMaterial pointer. Thus, skinning code could be moved from SkiinDeformer::Apply() to SkinDeformer::Update() and SkinDeformer::Update() could be called in the Scene::UpdateAnimation() loop. It does not matter if this update is called multiple times (e.g., once in Scene::UpdateAnimations() and again in the rasterizer) since it contains a guard to only do skinning if there has been a pose update since the last call to SkinDeformer::Update(). Since a pose can only be updated at most once per frame, the skinning step should happen at most once per frame regardless of how many times SkinDeformer::Update() is called.

It could be argued that the skinning code could have just be moved from SkinDeformer::Apply() to Scene::UpdateAnimations(), but this breaks the encapsulation of SkinDeformer, which is best avoided. The Scene does not need to know of the details of the mesh deformation caused by skinning, just that it needs to be performed.

The performance improvement from both the parallel pose updates and the parallel software skinning are shown in figure 9.

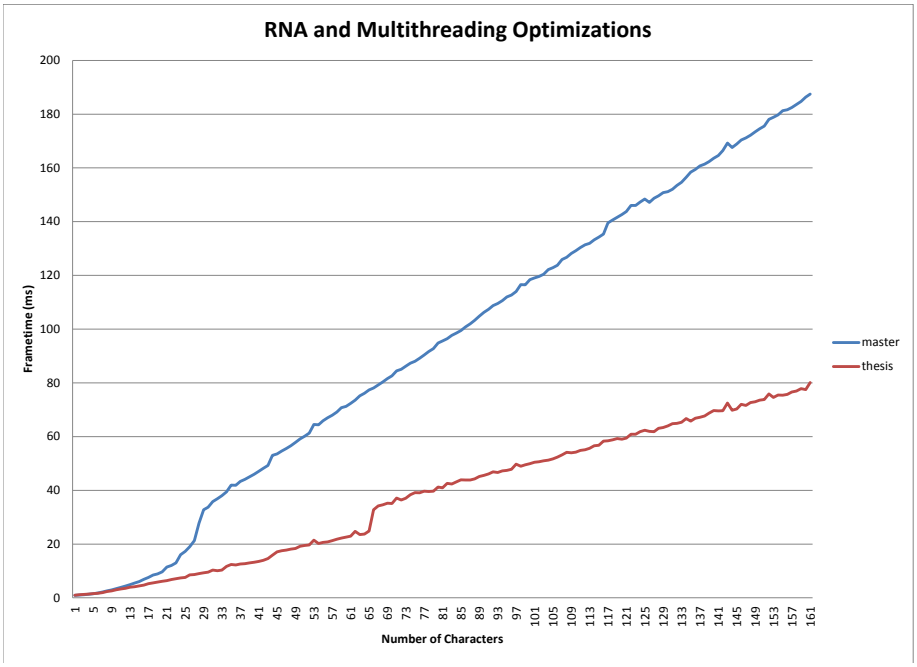

Figure 9: Results of RNA and multi-threading improvements

4.4 Hardware Skinning

To implement hardware skinning, a new vertex deformer type was added to the SkinDeformer, and each vertex deformer was made responsible for its own data. Previously, the Blender and BGE vertex deformers handled vertex data in similar ways, but hardware skinning does not need to copy data on the CPU before sending that data to the GPU. Currently the bone data is sent to the GPU as a uniform array of four-by-four transform matrices. New member functions were added to SkinDeformer to send the necessary uniform and attribute data to the graphics card. This allows the SkinDeformer to handle preparing the data, but allows the rasterizer to control when that data is sent to the GPU (i.e., when the OpenGL state is properly setup).

The modifications to Blender's vertex shader were based on NVIDIA code shown in listing 1. The modified vertex shader fragment can be found in listing 6.

Listing 6: Blender's vertex shader modified for hardware skinning

```
attribute vec4 weight;
attribute vec4 index;
attribute float numbones;

uniform bool useshwskin;
uniform  mat4 bonematrices[128];

varying vec3 varposition;
varying vec3 varnormal;

void hardware_skinning(in vec4 position, in vec3 normal,
                       out vec4 transpos, out vec3 transnorm)
{
    transpos = vec4(0.0);
    transnorm = vec3(0.0);

    vec4 curidx = index;
    vec4 curweight = weight;

    for (int i = 0; i < int(numbones); ++i) {
        mat4 m44 = bonematrices[int(curidx.x)];
```

```
        transpos += m44 * position * curweight.x;

        mat3 m33 = mat3(m44[0].xyz,
                m44[1].xyz,
                m44[2].xyz);

        transnorm += m33 * normal * curweight.x;

        curidx = curidx.yzwx;
        curweight = curweight.yzwx;
    }
}

void main()
{
    vec4 invertex;
    vec3 innormal;

    if (useshwskin) {
        hardware_skinning(gl_Vertex, gl_Normal, invertex, innormal);
    }
    else {
        invertex = gl_Vertex;
        innormal = gl_Normal;
    }

    vec4 co = gl_ModelViewMatrix * invertex;

    varposition = co.xyz;
    varnormal = normalize(gl_NormalMatrix * innormal);
    gl_Position = gl_ProjectionMatrix  * co;
```

The `usehwskin` uniform is always sent as zero. Any code wanting to make use of hardware skinning can later set it to one, which happens in SkinDeformer. Along with the `usehwskin` uniform, the SkinDeformer must handle the `bonematrices` uniform and the `weight`, `index` and `numbones` vertex attributes. The performance improvement for hardware skinning can be seen in figure 10

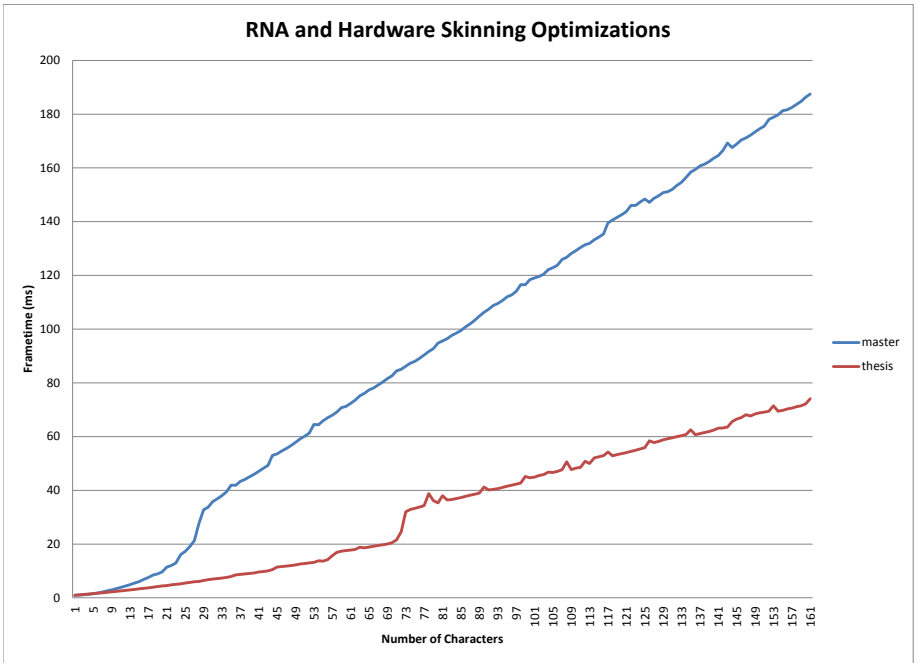

Figure 10: Results of RNA and hardware skinning improvements

4.4.1 Uniform Components and Bone Limitations

In OpenGL/GLSL, there is a hardware limit on the number of uniform components that a shader can use. A "component" is essentially a float, so every float sent as a uniform to a shader uses up a component. In order to leave components for the rest of Blender's vertex shader, the bone matrix data of the hardware skinning needs ensure that it does not use too many of the available components. To leave enough components for other parts of the vertex shader, an arbitrary limit is placed on the bone matrix data to only use half of the available components. Modern (shader model 4+) GPUs typically have 4096 components, which would give 2048 components for bone matrices, which allows for 128 matrices (a four-by-four matrix is sixteen components). Older GPUs tend to have 1024 components or less, giving a maximum of only 64 bone matrices on these graphics cards, but still leaves 512 components for other uniforms. 512 components is enough for Blender's current vertex shader and leaves ample room for future additions to the vertex shader. Further work could be done to find a smarter limit on components to squeeze more bones onto older GPUs.

Texture buffer objects (TBOs) could be used to work around component limits [3], but they are not supported on the older GPUs that would benefit from them, and 128 matrices is a reasonable limit on bones for the newer GPUs that support TBOs. TBOs could potentially be used to support unlimited (bounded by texture memory) bones, but then separate code would need to be maintained for hardware skinning on different GPUs.

4.4.2 Other Limitations

When using hardware skinning, each vertex can only have four bones influencing it. This limitation should work well enough in most cases. If a vertex has more than four bones influencing it, the four most influential bones are selected and their corresponding weights re-normalized (this is done in the SkinDeformer). Hardware skinning also requires a specific vertex shader, and as such only works with the BGE's GLSL material mode, and currently does not support custom shaders. If hardware skinning cannot be used (e.g., if the skeleton has too many bones), the skinning falls back to software skinning using the BGE vertex deformer.

5 Conclusion

With the benchmark scene described in section 3.3 running on the machine described in section 3.2, the optimizations presented give approximately a seven to eight times speedup. Overall, the optimizations presented show a greater performance improvement as more characters are added. A comparison of frame times between Blender's master Git branch and the branch used for this thesis work is presented in figure 11.

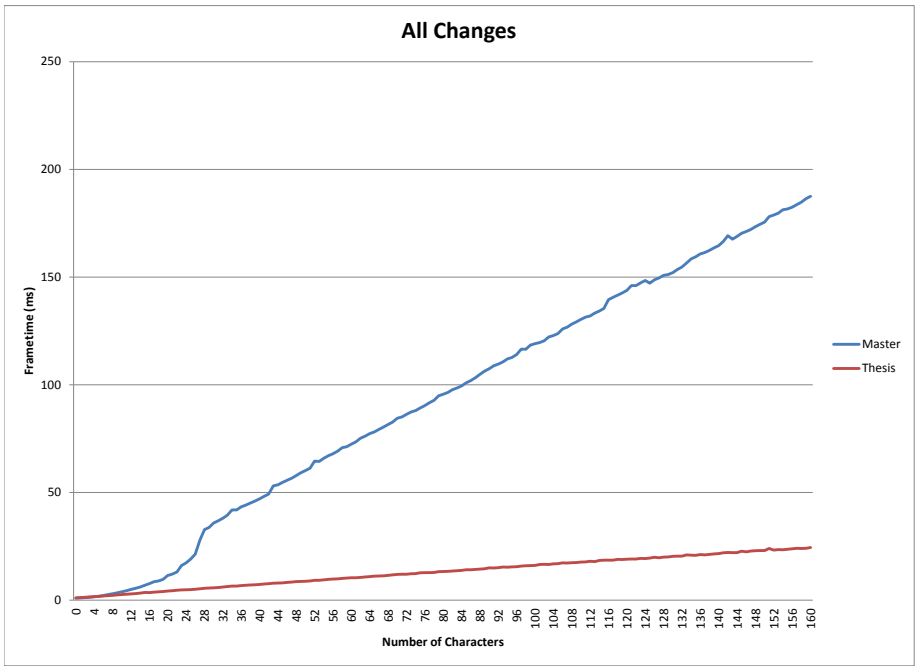

Figure 11: Comparison of frametimes between Blender's master branch and this work

On the test machine, the scene is CPU bound when the improvements are taken individually. However, with all of the improvements, the test scene becomes GPU bound. In order to further increase performance, the BGE's rendering system as a whole needs to be further optimized, which is outside the scope of this project.

References

[1] Blender Foundation, “Blender.” [Online]. Available: http://www.blender.org/

[2] “OGRE - Open Source 3D Graphics Engine.” [Online]. Available: http://www.ogre3d.org/

[3] The Khronos Group Inc., “ARB_texture_buffer_object.” [Online]. Available: https://www.opengl.org/registry/specs/ARB/texture_buffer_object.txt

[4] Unity Technologies, “Unity - Modeling Characters for Optimal Performance.” [Online]. Available: http://docs.unity3d.com/Documentation/Manual/ModelingOptimizedCharacters.html

[5] NVIDIA Corporation, “Nvidia Graphics SDK.” [Online]. Available: https://developer.nvidia.com/nvidia-graphics-sdk-11

[6] “Apache Subversion.” [Online]. Available: https://subversion.apache.org/

[7] “Git.” [Online]. Available: http://git-scm.com/

[8] B. VanLommel, “New Development Infrastructure,” 2013. [Online]. Available: http://code.blender.org/index.php/2013/11/new-development-infrastructure/

[9] “GitHub.” [Online]. Available: https://github.com/

[10] M. Stokes, “BGE Profile Stats and What They Mean,” 2012. [Online]. Available: https://mogurijin.wordpress.com/2012/01/03/bge-profile-stats-and-what-they-mean/

[11] “gperftools - Fast, multi-threaded malloc() and nifty performance analysis tools.” [Online]. Available: https://code.google.com/p/gperftools/

[12] NVIDIA Corporation, “NVIDIA DRIVERS Linux x64 (AMD64/EM64T) Display Driver.” [Online]. Available: http://www.nvidia.com/Download/driverResults.aspx/69372/

[13] “Valgrind: Tool Suite.” [Online]. Available: http://valgrind.org/info/tools.html

[14] Valgrind Developers, “Helgrind: a thread error detector.” [Online]. Available: http://valgrind.org/docs/manual/hg-manual.html

[15] ——, “Memcheck: a memory error detector.” [Online]. Available: http://valgrind.org/docs/manual/mc-manual.html

[16] “OpenMP.” [Online]. Available: http://openmp.org/

[17] “Big performance hog with OpenMP in VS2010 and hardware waits,” 2011. [Online]. Available: https://connect.microsoft.com/VisualStudio/feedback/details/640588

[18] S. Sharybin, “Task scheduler ported form Cycles to C,” 2013. [Online]. Available: http://lists.blender.org/pipermail/bf-blender-cvs/2013-October/059841.html

Vita

Author: Mitchell Stokes

Work Experience

2012 – Present Graduate Assistant Eastern Washington University

- Wrote software to assist in generating lab assignments for the database classes
- Wrote software to convert Oracle SQL to MySQL
- Assisted in overseeing and guiding Senior Project groups
- Taught Computer Literacy for four quarters

2010 – 2012 Computer Science Lab Assistant Eastern Washington University

- Assisted other students with their programming assignments
- Topics included introductory programming, data structures, and algorithms

Open Source Experience

Summer 2012 Student Developer Google Summer of Code

- Implemented asynchronous asset loading
- Improved the performance of converting Blender data to game engine data
- Implemented a Pre-Z pass optimization to minimize the effects of overdraw
- Closed approximately sixty-five bug reports

Summer 2011 Student Developer Google Summer of Code

- Overhauled the Blender Game Engine's animation system
- Added features such as animation layers and an animation Python API

Summer 2010 Student Developer Google Summer of Code

- Worked on better custom shader support for the Blender Game Engine
- Allowed custom shaders to be displayed in Blender's viewport (outside of the engine)
- Added support for geometry shaders
- Added an improved Python API for custom shaders

2008 – Present Developer Blender

- Gained commit rights in 2009
- Listed as a top 20 Blender developer of 2013
- Help to maintain and develop the Blender Game Engine (includes rendering, animation, physics, Python scripting API)

Projects

- *Bgui*, a Python GUI library for the Blender Game Engine