Priority based level of detail approach for interpolated animations of articulated models

BY

Antoine Abi Chakra

Bachelors in Computer Sciences, DigiPen Institute of Technology, 2006

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science
in the graduate studies program
of DigiPen Institute of Technology
Redmond, Washington

United States of America

Summer

2008

Thesis Advisor: Dr. Xin Li

DIGIPEN INSTITUTE OF TECHNOLOGY

GRADUATE STUDY PROGRAM

DEFENSE OF THESIS

THE UNDERSIGNED VERIFY THAT THE FINAL ORAL DEFENSE OF THE MASTER OF SCIENCE THESIS OF ANTOINE ABI CHAKRA

HAS BEEN SUCCESSFULLY COMPLETED ON JULY 31ST 2008

TITLE OF THESIS: ANIMATION LEVEL OF DETAIL

MAJOR FIELD OF STUDY: COMPUTER SCIENCE.

COMMITTEE:

Dr. Xin Li, Chair	Dr. Matt Klassen
Samir Abou Samra	Dr. Jason Hanson

APPROVED :

[signature] [date]	[signature] [date]
[name] date	[name] date
Graduate Program Director	Associate Dean
[signature] [date]	[signature] [date]
Dr. Xin Li date	[name] date
Department of Computer Science	Dean

The material presented within this document does not necessarily reflect the opinion of the Committee, the Graduate Study Program, or DigiPen Institute of Technology.

INSTITUTE OF DIGIPEN INSTITUTE OF TECHNOLOGY

PROGRAM OF MASTER'S DEGREE

THESIS APPROVAL

DATE: _____[DATE]_____

BASED ON THE CANDIDATE'S SUCCESSFUL ORAL DEFENSE, IT IS RECOMMENDED THAT THE THESIS PREPARED BY

[Antoine Abi Chakra]

ENTITLED

[Animation Level of Detail]

BE ACCEPTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF MASTER OF COMPUTER SCIENCE FROM THE PROGRAM OF MASTER'S DEGREE AT DIGIPEN INSTITUTE OF TECHNOLOGY.

[Signature]

[Name]

Thesis Advisory Committee Chair

[Signature]

[Name]

Director of Graduate Study Program

[Signature]

[Name], Associate Dean

[Signature]

[Name], Dean of Faculty

The material presented within this document does not necessarily reflect the opinion of the Committee, the Graduate Study Program, or DigiPen Institute of Technology.

14 Future Work

15 References

Abstract

Level-of-detail techniques are widely used in games to lower the computational cost for model animation, object rendering and
artificial intelligence. This paper presents a level of detail technique targeting interpolated animations of articulated characters. It assigns a priority for animation key frames at each bone in the model hierarchy. Some heuristics rules are proposed to measure the "importance", based on a collection of information such as the greatest translation value, the greatest rotation value, the greatest scale value, the greatest number of children per bone, the greatest depth of the hierarchy, etc. At run time, the distance separating the animated model from the viewer is used to scale the pre-calculated priority. The value is then normalized and used by the animation controller to reduce a percentage of transformation interpolations.

1 Introduction

With the transition from 2D to 3D, computer hardware was (and still is) being pushed more and more to produce better picture quality. Each generation of graphic cards adds new features to its predecessors, new techniques are easier or faster to implement, and the goal remains to same: Squeezing more and more juice power into 3D applications. If programmers relied only on hardware power, then we would be forced to wait for the next generation of graphic cards to get faster frame rates and better looking games and 3D applications. Fortunately, that is not the case, because software optimizations are always implemented in order to alleviate hardware processing. A basic example of software optimization is frustum culling: Instead of sending all our scene's objects to hardware processing, we first preform a relatively quick check to see if each object is inside the viewing frustum, or in other words if it's visible by the viewer. If it's not visible, then we don't send it to hardware processing (Shading, special effects, rasterizing...). This simple optimization will drastically decrease the number of processed polygons, thus reducing the amount of time to process our 3D scene, which will allow us to implement more special effects.

The problem with most optimization techniques is that they all look at our problem from the same perspective: Reducing the number of polygons sent to hardware processing. But what about processed polygons? What can we do about objects that are visible by the viewer? Here's where “Level of Detail” comes into action. Unlike other optimization methods, Level of Detail deals with visible polygons rather than unseen polygons. After performing all other “polygon reduction” techniques, we are obviously left with the visible objects of the current frame. How can we reduce the polygons of these objects? Let's say an object of more than 50,000 polygons is visible by the viewer started to move back and back until it becomes really small on the screen (but still visible), do we still need to process 50,000 polygons? Many of these polygons will overlap, thousands of pixel will end up on the same screen position, overwriting the previous color over and over again. That seems like a huge waste of processing time, which could have been

devoted to implement more special effects and eye candy. Level of Detail takes the distance (or screen space, more on that later) that separates a visible model from the camera into consideration before rendering it. If that distance is great, we should not process all the object's polygons. On the other hands, if that distance is relatively small (object very close to the viewer), we should render each polygon, because the removal of even a single polygon could be noticed by the user.
There are two main types of Level of Detail techniques:

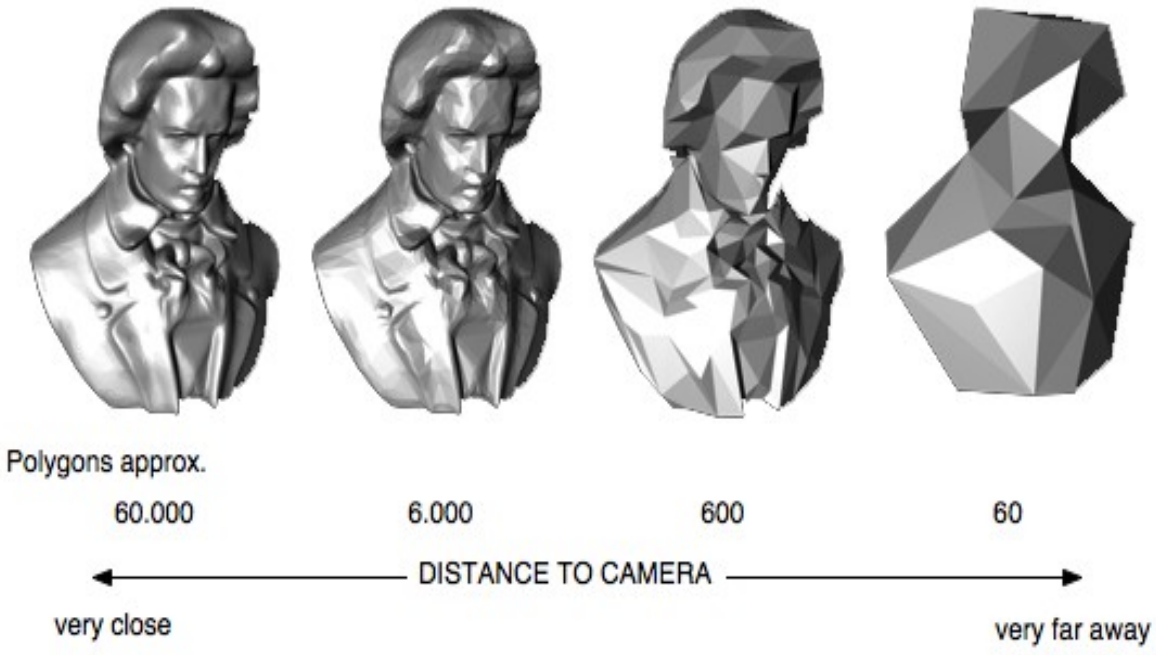
- Discrete LOD
- Continuous LOD

2 Discrete LOD

Discrete LOD reduces the number of polygons of 3D models in a discontinuous way. The artist is responsible of creating several models for each object, each with a different number of polygons. Although these models differ drastically in their polygon count, they should try their best to preserve the original shape. At run-time, a different model is rendered each frame depending on the distance separating the object from the camera. In other words, if the object is very close to the viewer, the most detailed model is rendered, while on the other hand, if the object is far, a less detailed version is chosen.

The advantages of using discrete LOD is that it basically doesn't add any computations at run times, besides calculating the distance that separates the object from the view point. It's also easy ti implement, it doesn't require vertex or polygon management, since all what is done is switching from one model to another. But these advantages come with a price. Each object now will have several meshes attached to it, which will drastically increase the needed amount of storage memory. Another disadvantage of using this simple method is that the

silhouette of the object is preserved, since we can't create a simplified mesh for all possible view angles, therefore the artist's simplification will be uniform across the entire mesh. Last by not least, we should take special care when switching from one model to another in order to avoid the disturbing popping effect, which can be very annoying if switching was poorly executed (Hard mesh switching)



3 Continuous LOD

Continuous LOD deals with the popping problem which occurs with discrete LOD. Look at the previous example. When switching from the 60,000 polygon model to the 6,000 one, a disturbing popping effect will occur due to the great geometric difference between the two models, which will also lead many other differences like shading differences. Instead of having several static meshes where one is chosen depending on the distance between the object itself and the view point, continuous LOD suggests using the original mesh only (the most detailed one, or in other words the one with the highest polygon count), and decrease its polygon count gradually as it gets smaller and smaller on the

projected plane. It's also worth mentioning that this method greatly reduces (if not removes) the amount of required memory, because unlike the previous method (Discrete level of detail), this method doesn't require any additional meshes and data.
But this advantage doesn't come for free. We have mentioned that in order to switch to a lower detail mesh, we will have to gradually decrease its polygon count as the object gets further from the view point. This “polygon elimination” requires a thorough analysis of the mesh, because the general visual shape of the mesh will be much more affected by the removal of some “important” polygons compared to the other polygons.
Take a cone for example. If you remove just the head vertex, the entire shape will dramatically be altered! (As a matter of fact, you won't have a cone anymore!). But if you remove a vertex from its base surface, you will still have the general shape of a cone.
In order to know which polygons to remove at run time, we will have to build a hierarchy tree at preprocessing time which will hold the precise order of polygon removal. Note that this tree will be traversed in both direction, because if the object is getting closer to the view point, we will have to do the opposite process, which is adding previously removed polygons to the mesh, or in other words increasing its detail. All this polygon removal/addition is executed using local simplification operators.

4 Local Simplification operators

Continuous Level of Detail relies on local simplification operators in order to decrease the number of polygons of a mesh at run-time. The choice of which simplification operator to use is highly dependent on the mesh's complexity and triangle formation. Note that it's not only better to choose a simplification operator over the other for a certain situation, but sometimes it's highly important to avoid using one of them in a situation where a deformation could occur, thus greatly changing the shape of the mesh in question, which could lead to artifacts at run-time.

4.1 Edge collapse

This is one of the mostly used local simplification operators. One of its main advantages is its constant run-time cost, which is greatly appreciated by programmers who try their best to maintain a constant frame-rate in real time applications.
How this operator works is a bit straightforward, and it can be summarized: Two vertices are replaced by one. Of course, the two chosen vertices must meet some conditions before this simplification operator can be applied to them. The first condition is that these two vertices must be neighbors, or in other words representing an edge. The second condition will be explained later on. So as we said, two vertices representing and edge are removed and replaced by one, but where is the new vertex positioned?
Well there are two ways to resolve that issue. The first method is done by positioning the new vertex in one of the removed vertices' positions, also called a half-edge collapse, while the second method is done by completely creating a new vertex and positioning it somewhere between the two removed vertices, or on the removed edge in other words, also called a full-edge collapse.
It's clear and safe to say that the first positioning method is less expensive computation wise, since it doesn't involve creating a new vertex, but actually removing one since the other one (which wasn't removed) will be used as the new vertex. But on the other hand, the second method allows a more flexible simplification because the new vertex can be positioned according to vertex weights. What that means is that sometimes in meshes, some vertices have more importance than others, and the removal of these vertices will affect the mesh's shape more drastically compared to removing other vertices. Therefore, if one of these important vertices is to be removed, it's better to position the newly created vertex as closely as possible to the most important vertex.
After performing this edge collapse operator, some “connection” tasks must be performed in order to maintain the mesh's continuity. What happens to all the edges that connected to one of the two removed vertices? Well the answer to that question is simple: All edges

that connected to any of the removed vertices will connect to the new vertex (the one that replaced the previous two).
Although this method seems bullet-proof, it may generate some problems which will later generate artifacts at run-time. One of the main problems that can occur when performing this simplification operator is to create what is called a “Mesh-Foldover”. What this means is that since many faces will be affected by this operation, an affected face might hide a part of a previously visible face. Let's say face A and face B are both visible from a certain view-point. After applying the “edge-collapse” operator on an edge close or directly connected to these two faces, we will have to adjust all the edges that connected to any of the two removed vertices, thus changing some of the triangles formation. So, face B might hide or cut through face A, which will, as mentioned earlier, generate annoying artifacts at run-time.
One way to avoid having this mesh foldover problem is by comparing the previous and current normals of all the affected faces. If on normal changes by more than 90 degrees, then it's highly probable that its respective face is now hiding or cutting through a previously visible face, which means that we shouldn't apply this simplification operator on the current edge.
Another problem with the edge collapse operator is that it might generate some topological inconsistency. This happens when the neighbors of the two removed vertices are more than three. In other words, if the two ends of the edge we're going to remove have more than three common neighbors (Vertices), then it's highly probable that the new mesh will have some topological inconsistency, thus it's better to avoid applying this simplification operator of the edge in question.

4.2 Vertex-pair collapse

This simplification operator resembles a bit the previously explained operator (Edge Collapse). In fact, it's also called the “virtual edge collapse”. The main difference

between the vertex-pair collapse and the edge collapse is that the two removed vertices do not represent an edge in the vertex-pair collapse operator. So what this operator does is that it removes two unconnected vertices and and replaces them by one. The position of the new vertex can be chosen similarly to how we chose it for the previous operator; in other words, it can be one of the two removed vertices, which is the less expensive way computation wise, or it can be positioned somewhere between those two, depending on the vertices weights.
Another question rises up which is which two unconnected vertices to choose? Let's say we have a mesh of n vertices. This gives us a a potential of n^2 virtual edges to choose from. Of course, applying the virtual edge collapse on all these virtual edges then choosing which is the best to go for is highly expensive computation wise, therefore, we should only choose a virtual edge where its two vertices are close neighbors.
Also, similarly to the previous simplification operator, all edges that connected to one of the two removed vertices must be now connected to the newly created vertex in order to ensure the mesh's continuity.

4.3 Triangle collapse

This simplification operator has a greater effect on the mesh since it doesn't apply to a single vertex of edge, but to a whole triangle. What this operator does is that it removes a triangle and replaces it by a single vertex, thus decreasing the total number of triangles, which is our main goal in this level of detail subject. This operator has a greater effect on the mesh since it removes two vertices instead of one compared to the two previously mentioned simplification operators. Again, we will have to make a choice here on where to position the new vertex. As we did previously, we can choose the faster method by choosing on the three removed vertices to represent the new vertex, or we could create a new one and positioning it most probably somewhere inside the removed triangle, according to vertices' weights and importance.

So how to adjust all the neighboring edges after applying this operator? Similarly to previous methods, we will have to link back all the edges that connected to any of the three removed vertices to the newly created one, thus preserving the mesh's continuity. Note that this method is equivalent to two edge collapses, since it removed two vertices in one step, while the edge collapse operator removed one vertex in one step.

4.4 Cell collapse

This simplification operator is a bit different than previous operators, and isn't a general method to simplify a mesh. As it's title clearly says, this operator projects a mesh onto a grid, then it simplifies it according to some pre-specified grid rules that can be set by the user. One of the most general rules is to perform a vertex union for all the vertices that were projected onto the same grid cell. What that means is that after projecting the mesh onto a grid, one or more vertices may end up in the same grid cell. So what this rule does is that it makes sure that each grid cell won't contain more than one vertex. As you may be imagining, this method can drastically decrease the number of triangles of a mesh, especially if many vertices end up sharing the same grid cell.
After applying this simplification operator, we will obviously have to link back all the edges. This requires some extra steps compared to the previously explained simplification operators. After projecting the mesh onto the grid, we will have to keep record of all the connected grid cells. What this means is that if a vertex in cell (2, 3) connected to a vertex in cell (5, 1), then after the vertex union has been made for all cells, we will have to connect back the new vertex in cell (2, 3) to the new vertex of cell (5, 1), thus making sure that the new simplified mesh resembles as much as possible the previous high polygon mesh.
Although this simplification operator seems efficient, it must be said that there are a lot of situations where the new simplified mesh looks like as if it is missing a lot of parts, and not actually representing a simplified mesh of the original one. This is highly dependent

on the grid's resolution. The more cells we have, the more accurate the new mesh will be, but on the other hand the difference in the number of polygons won't be great. On the other hand, if the grid had few cells (low resolution), the number of polygons will decrease drastically, but the new mesh's fidelity to the original one will be poor.
One other problem with this method is that it's very expensive to perform each game loop. Let's say we have a model of then 10,000 polygons. Can you really imagine projecting then thousand polygons onto a grid each game loop? That would certainly decrease our frame rate, which will negate our level of detail benefits! Therefore as mentioned in the beginning of this paragraph, this simplification operator can't be applied to any mesh, and it's generally used to simplify terrains. What is the main difference between a terrain mesh and a general random mesh? Well as you may know, a terrain mesh is made from an array of height for each vertex, thus in a way representing a grid by itself, which makes it easier to project on another grid, since the computation time for this projection is cheap because it only involves setting the vertex height value to zero.

4.5 Vertex Removal

This simplification operator is of the least used operators due to its high computation time. What it does is that it completely removes a vertex from the mesh. The main problem here is that by removing a vertex, an area of more than three vertices will be created, which will be rejected or at least cause problems during the whole pipeline, since all these process are made to handle polygons of three vertices, or in other words triangles! So we will have to do after removing a vertex is triangulate the newly created area, thus making sure that we won't have any polygon consisting of more than three vertices. These frequent triangulation calls are the main reason why this simplification operator is usually avoided, because any programmer knows the high computation cost of the triangulation function, which is called once each time this operation is applied.

4.6 Polygon merging

This operator can be somehow considered as an optimized version of the vertex removal local simplification operator. What this operator does is that it merges several nearby polygons into a single one. It's also important to point out here is it's better that the combined polygons have similar or close normals, which will make sure that effects like lighting won't change drastically when this operation is performed.
So what does “merging polygons” mean? Well let's say we have two nearby triangles. If we remove one of their common vertices or edges, we will be merging two distinct areas into a single one. Therefore, this method removes several vertices in a single step, then it calls the triangulation function in order to triangulate the newly created area. The main optimization here compared the the “vertex removal” operator is that we are removing more than one vertex before calling the costly triangulation calls, which is obviously way faster than calling it for each removed vertex.
One of the main concerns about using this simplification operators is that if not properly used, it might generate some holes in our mesh! To avoid this problem, it's better not to remove a great number of vertices before calling the triangulation function, and to try choosing faces with similar normals before merging them.

5 Local simplification operators analysis and comparison

5.1 Half edge collapse

This operator requires few run time calculations compared to other operators, but it doesn't provide us with a high fidelity mesh compared to its full edge collapse counterpart. It doesn't require any allocation at run time since it doesn't have to create any new vertices, because the vertex replacing the removed two vertices is one of them,

which on the other hand reduces the number of affected triangles. In other words, only triangles that connected to the collapsed vertex will be affected. The main advantage of using this simplification operator is that the vertices of any simplified mesh represent a subset of the original full detailed mesh, because we're not changing the position of any vertex during this process.

5.2 Full edge collapse

This operator generates a great flexibility which provides us with a high fidelity simplified mesh since the position of the new vertex replaced the two removed vertices can be chosen according to vertices weights in order to preserve the mesh overall appearance. On the other hand, this advantage comes at price, which is performing some extra calculations in order to determine the position of the new vertex. Another disadvantage is that the vertices of a simplified mesh are totally or partially different than the ones of the original one, since each time we apply this operator, a new vertex is created and used.

5.3 Cell collapse

This operator isn't performed on a single edge or vertex like other operators, but instead on the whole mesh. It has a near constant time which is relative to the number of vertices of the mesh in question, since we will have to project all these vertices onto a grid (Explained previously). On the other hand, this method can't be used for any mesh, since the rotation of the object can drastically affect the projection of its vertices, and is therefore only used for flat-like meshes likes terrains, where the projection of vertices is simulated by just setting their height values to zeros.

5.4 Vertex removal

This method generates a high fidelity simplified mesh, since we could use the vertices weights and remove the one with the lowest importance value in order not to affect the original mesh's appearance. On the other hand, it requires some heavy calculations because of all the triangulations, since for each removed vertex, a triangulation function call is needed to triangulate the new area.

6 Deciding when to switch between LODs

We have talked so far about different ways to implement a level of detail engine into a 3D application, which are the use of several pre-modeled meshes that vary in their number of polygons, their texture maps and so on, or the use of progressive meshes which gradually decrease the number of polygons in a mesh by using one or several local simplification operators like the edge collapse or the vertex removal. What we will talking about now is how to decide when it's the right time to switch between two discrete LODs. How to know that the currently used mesh is “over-detailed” and it should be replaced with a less detailed mesh? There are several ways to handle that issue, and of course, each one has its advantages and disadvantages.

So far, we know that our main goal is to reduce the number of polygons as long as their existence doesn't contribute to the scene, which means when the object using that mesh becomes very distant from the viewer, or in other words very small on the screen. Different “switch-handling” approaches will be discussed and compared in order to determine which method is best suited for which situation.

6.1 Distance

6.1.1 Description

This is the most used method to switch between different discrete LODs. Since the distance between the object and the viewer is directly related to how big or how small this object is perceived, then it could be used as a decision factor for LOD switching. A distance interval has to be set for each LOD from the most to the least detailed one, and each game loop, we check the distance that separates the object from the viewer, or in other words the active camera, and finally we'll know to which interval does this distance belong to, which will directly provide us with the appropriate LOD to use.

Here's how the structure for a discrete LOD would look like:

```
struct Discrete_LOD_Distance
{
	list<Mesh *> allMeshes;
	list<float> distanceIntervals;
	int current;
};
```

The first member of this structure contains pointers to all the meshes that will be used by the current object, from the most to the least detailed one. This means that “allMeshes[0]” is a pointer to the mesh that should be used when the object is very close to the active camera. The “distanceIntervals” list will contains several incrementing distances which will be used in parallel with the previous list, in other words, each mesh in the “allMeshes” list has a correspondent interval in the “distanceIntervals” list. Finally, current will be used to save which mesh should be currently used, so that we render it when the render function of our object is called.

The structure of the continuous LOD is a bit different:

```
struct Continuous_LOD_Distance
{
	Mesh *myMesh;
	float ratio;
	float minDistance, maxDistance;
	int CurrentNumberOfSimplifications;
};
```

The main difference between the continuous LOD structure and the discrete one is the existence of only one mesh in the continuous LOD structure, since simplifications will be done on one mesh (using local simplification operators) in order to decrease the number of polygons. The existence of only two distance boundaries is another main difference. For this type of level of detail, we will set the minimum and maximum distance where simplifications will be made to our mesh. If the distance if less than our pre-set minimum distance, then the full unchanged mesh will be rendered. If the distance belongs somewhere between the distance interval (which is bounded by minDistance and maxDistance) a certain number of simplifications will be done before rendering the mesh. Finally, if the distance is greater than “maxDistance”, then no further simplifications should be executed, and the last simplified mesh will be used to render the object.

So how to determine the number of simplifications that should be done on the mesh using the distance that separates it from the viewer? A certain ratio should be calculated using the distance boundaries and a certain minimum number of polygons. Let's say we have a mesh of 10,000 polygons, and we decide that we don't want it to be simplified to less than 2,000 polygons. We set the minDistance and maxDistance variables to any logical values. Let's assume that minDistance was set to 100 and maxDistance to 1,000.

So far, we know that if the distance is less than 100, the full 10,000-polygon-mesh will be displayed, and if the distance is greater than 1,000, our mesh will have to be simplified to 2,000 polygons before we render it. But what if the distance is somewhere inside that interval? We will have to compute a ratio using that range's lower and upper bound, and use it in order to determine how many polygons the mesh should be simplified into.

We kept mentioning that in order to determine which mesh to render or the number of simplification to do, we will have to calculate the distance that separates the object from the active camera. But as we all know, our object is not a single point! It's a collection of adjacent triangles. So the inevitable question rises here: Which point to use in order to determine that distance. There are several approaches that could be used to handle this problem.

6.1.2 How to choose the reference point?

6.1.2.1 Center of the object

The first solution is to use the center of the object as the point to use when the calculating the distance between the camera and the object. This is very easy to implement, since the center of the object is usually (0, 0, 0), therefore the world position of the object's center will be the actual position of this object in the world. This method seems good enough, but what if the object's weight is somehow far from its center? In other words, maybe the real dense part of the object (The part that contains most of the polygons) is a bit far from the center? Should we wait till the center of the object passes the distance threshold before we switch to lower detailed LOD? This is one of the main problems with using the object's center to represent the whole object when calculating the distance.

6.1.2.2 Point chosen by the artist

Using a point chosen by the designer or the artist is one of the solutions for object having their polygon weight away from their center, or in other words, their center of mass drastically different that their center. It requires some extra calculation compared to the previous method, since we will have to translate and rotate the chosen point in order to determine its world position, compared to only translating the center previously. This method guarantees that the distance between the object and the camera represents the distance between the viewer and the dense part of the object, since this is what we really need in order to determine if a simplification should be done or not. On the other hand, this method has a major disadvantage, which is having a very noticeable popping effect especially if the object is rotating around the threshold area.

6.1.2.3 Closest point to the viewer

The third method to determine the distance that separates the object from the active camera is to use to closest point to the viewer each time we want to calculate this distance. As you may have noticed, this is the most complex method computation wise, since calculating the closest point to the viewer is not a simple task, although it could simplified to just using the closest bounding box point. But does this method guarantee the removal of the popping effect if the object is rotating around the threshold area? Unfortunately it doesn't, since the closest point can be going back and forth through the threshold distance while the object is rotating.

Using distance to determine which LOD to use is simple to implement, it's also efficient since only few conditional statements are enough to determine if a LOD switch is necessary. On the other hand, one of its main disadvantages is that only one point will be

used to represent the whole object while calculating the distance. Some optimizations should mentioned about this method is that although we have to calculate the distance between the object and the camera, we could switch that to calculating the square of the distance, which allows us to avoid the use of the expensive square root function. Few modifications should be done on our structures, like replacing the “distances interval” by a “square distances intervals” for the discrete LOD strcuture, and the “min/max distances” by “min/max square distances” for the continuous LOD structure. Another optimization can be done is to avoid doing this check each game loop. As game programmers know, game objects don't really move at drastic speeds during the game, therefore checking if a LOD switch is needed every two or three game loops can be easily done to reduce unneeded calculations.

6.2 Size

6.2.1 Description

This is another widely used to determine if a LOD switch is needed. Instead of using the viewer-object distance in the world coordinates system, we use the size in the screen coordinates system. Similarly to the previous method, switching between different discrete LODs or determining the number of needed simplifications is based on the actual size of the object on the screen, or in other words, on how it is really perceived by the player.

This method has some great advantages over the distance method, where the most important one is that all the object contributes to the determination of the LOD switching instead of only one point. As we mentioned in the previous method, we relied only one one point to determine the distance between the active camera and the object in question. Although there are several ways to choose this particular point, we were still facing the

same problem of using one point to represent the entire object. Another advantage of using the screen size of the object is that scaling the object will have a direct effect on our LOD decisions. Let's take this example: Object A is 500 unit away from the viewer, and we are using its center to determine its distance from the camera. Now assume that object A is scaled by a factor of 3, in other words, its size on the screen has tripled. What is the distance between its center and the viewer? It's still 500 because the center of the object isn't affected by its scaling. Should we stick to the mesh that was used before we scaled our object? Or should we switch to a more detailed one because the object is bigger on our screen , which means every bit of extra detail will be noticed by the viewer? To get back to our list of advantages, we can safely say that using screen size to determine which LOD to use is directly dependent on the object's scaling.

The structures for this method are similar to the ones of the distance method; all we have to do is switch each distance values by a size value.

```
struct discrete_LOD_Size
{
	list<Mesh *> allMeshes;
	list<float> sizeIntervals;
	int current;
};
```

The first member of this structure contains pointers to all the meshes that will be used by the current object, from the most to the least detailed one. This means that “allMeshes[0]” is a pointer to the mesh that should be used when the object occupies a relatively big area on the screen. The “sizeIntervals” list will contains several incrementing sizes which will be used in parallel with the previous list, in other words, each mesh in the “allMeshes” list has a correspondent interval in the “sizeIntervals” list. Finally, current will be used to save which mesh should be currently used, so that we render it when the render function of our object is called.

This is the structure of the continuous LOD:

```
struct Continuous_LOD_Size
{
	Mesh *myMesh;
	float ratio;
	float minSize, maxSize;
	int CurrentNumberOfSimplifications;
};
```

Again, we only need one mesh for the continuous LOD structure, since simplifications will be done on one mesh (using local simplification operators) in order to decrease the number of polygons. For this type of level of detail, we will set the minimum and maximum size where simplifications will be made to our mesh. If the size if less than our pre-set minimum size, then the full unchanged mesh will be rendered. If the size belongs somewhere between the size interval (which is bounded by minSize and maxSize) a certain number of simplifications will be done before rendering the mesh. Finally, if the size is greater than “maxSize”, then no further simplifications should be executed, and the last simplified mesh will be used to render the object.

On problem remains which is how to compute the size of a certain object in the screen space before rendering it? Remember that knowing the size occupied by the object on our screen before its render call is crucial because we will have to set its needed level of detail at pre render time. There are several ways to determine this size value, they be explained and compared in the next chapter.

There are two main methods for determining the approximate size occupied by a certain object on the screen.

- Axis aligned bounding box
- Bounding sphere

6.2.2 AABB (Axis aligned bounding box) projection

In order to determine the screen size occupied by a certain object using this method, we will have to project its axis aligned bounding box (AABB) onto the screen space. This id done by determining the eight boundaries of the AABB in the object coordinates system, and transform them to the screen coordinates system by multiplying them by all the matrices (world, view, perspective) in the correct order. After doing so, we will get a 2D rectangle or trapeze whose area is the size we are looking for.

This method is a bit expensive computation wise, since it requires 8 matrix-vertex multiplications, then an area computation of a random trapeze. Another main disadvantage is that its very rotation dependent, which could force our application to switch between different LOD if an object having its width, height and depth sizes very different is rotating.

6.2.3 BS (Bounding sphere) projection

This is another method to approximate the screen of an object in screen space. Instead of projecting the object's AABB to screen space, we project its bounding sphere (BS). This is accomplished by projecting two points from the object's coordinates system to the screen coordinates system: The center of the sphere (A) and a point from the sphere (B), where the vector AB is parallel to the camera's right vector. In other words we are projecting the radius of that bounding sphere to screen space. After calculating the radius in screen space, we will be able to determine the area occupied by a circle having this

radius. This method is cheaper than the previous one, since it only requires the two matrix-vertex multiplications instead of eight, and it offers a real advantage, which is being rotation invariant, because bounding spheres are not affected by the object's rotation. On the other hand, bounding spheres usually offer a poor fit for meshes, and programmers avoid using implementing them since they have a poor use especially in collision detection engine.

Using the screen size of an object to determine if a level of detail switch is needed is generally more expensive than using the distance criteria, since it requires at least two matrix-vertex multiplications (in case the bounding sphere is used) or eight matrix-vertex multiplications (in case the axis aligned bounding box is used), but on the other hand, it offers some great advantages like taking the object's scaling into consideration, and using the entire object to determine the needed level of detail instead of only using one point.

6.3 Priority

6.3.1 Description

This level of detail switching criteria is a bit different than the previous two, because it doesn't actually rely on measures to compare or evaluate the current visibility of objects, but instead it relies heavily on human intervention. The idea of using priority in games to determine the needed level of detail came from the fact that in most if not all games, players will be focusing on some object way more than other objects. For example, let's take a soccer game. I think it's safe to say that during the majority of a match, people will be focusing mostly on the ball, which will always be visible in the viewport, then comes the soccer players, who might not always be the center of attention depending on their field position, and finally, we have all the objects around the soccer field like spectators, flags etc...

In conclusion, we can say that in every computer game, some objects have greater importance in presenting the scene compared to others, and a level of detail system uses that facts to decide which objects should be degraded.

What we will have to do here is to assign a priority ranking for each game object. Most important objects have the highest rankings and the least important objects have the lowest ones. Let's assume that our game programmers have set a maximum of 60,000 polygons per scene. Now during gameplay, if a scene surpassed this threshold number of polygons, we will have to start switching some objects to their lower level of detail models, in order to get back to our 60,000 polygons limit. Thanks to our priority list, and of course taking into consideration which objects are currently visible on our screen (because lowering the detail level on objects that are not currently visible in the scene won't have any effect), we can start lowering the level of detail of objects starting from the least important objects, hoping that we will reach our threshold of 60,000 polygons before having the degrade the most important objects in the scene.

This method gives programmers another indirect tool, which is the ability to save some objects from being degraded, by simply not putting them on the priority list! For example, if you are programming a game and you are implementing a level of detail engine using the priority rankings, and you have some objects that you always want to display in full detail, then just don't add them to the list! As simple as that. Since your level of detail engine will only handle or affect objects added to your priority list, objects not belonging to it will be safe from being degraded to their lower level of details.

It's also worth mentioning that this priority list doesn't have to be static during the whole game, and that fact applies to object's rankings as well. If you know what you are doing, and depending on the game status, you can manipulate which objects are currently on the priority list and which ones are not, or maybe change the priority ranking of some objects depending if their current importance to the scene has increased or decreased.

6.4 Perceptual factors

This level of detail method uses the fact during some games, players will focus on certain objects more than other in certain situations. Take a first person shooter for example. During an intense battle, you can be sure that the player will be focusing on the enemies he or she is shooting on, or at least looking at the areas where these enemies have vanished to, since they might pop up from the same place they disappeared from. One of the other perceptual facts in games is that humans can perceive a clear image of objects moving at high speeds, especially if these objects are crossing the screen in a relatively short amount of time.

What this LOD switching method does is that it determines which objects are currently "poorly perceived", or at least "less-perceived" compared to others, and switch these objects to their lower level of detail in order to decrease the overall number of polygons in the scene.

As you may have noticed, this method is a bit hard to implement because it doesn't rely on concrete values to compare objects among themselves, but rather on the human perceptual factors which are by no way measurable, not to forget that each person's perception vary and can be very different compared to others.

6.5 Level of detail switching criteria comparison

We covered four different strategies for determining which objects should be degraded, or in other words switched to a lower level of detail, and under which circumstances should any object be saved from visual degradation. The first method relies on the distance that separates the object from the active camera in order to determine the

required level of detail, which is logical since the greater that distance is, the smaller the object will appear, which allows us to reduce its level of detail without degrading the scene visually. Although this method seems logical and straightforward, and has its flaws, where one of them is mainly not being affected by the object's scaling, which directly affects the final size perceived by the viewer. The second method, which relies on the approximation of the final size of the object on the screen solves the first method's problem, since we are directly dealing with the core issue, which is knowing the number of pixels this object is going to occupy before rendering it. But of course, this extreme piece of information comes at a price, since this method requires lots of calculation especially several matrix-vertex multiplications. The last two methods which are the priority ranking and the perceptual factors are - especially the latter one - somehow less concrete than the previous methods, since they don't rely on actual mathematical calculations to determine if a level of detail switch is needed, and they are usually avoided and replaced by the distance or size methods, since they allow the game engine to be stable and avoid having run time surprises.

7 Avoiding poppups in discrete LOD switching

Using discrete level of detail is the most used method, and it had gained it popularity over the continuous level of detail because of its simplicity, stability and efficiency. The main advantage of using discrete level of detail is that you will know exactly what you will know. If you set the distance or size intervals properly, that is from lowest to highest (in case you are using the distance criteria) or from smallest to greatest (in case you are using the size criteria), you will see your objects switching from a mesh to another as they pass through the threshold. On the other hand, using a continuous level of detail engine will make drastic changes to your mesh, using local simplification operators, and programmers must take extreme care of the order of simplification of each mesh, since each simplification pass can lead to a different simplified mesh. On the other hand, using

discrete level of detail will force artists to draw more than one mesh for each model, which won't only take extra production time, but will severely increase the required storage memory. This problem doesn't exist if continuous level of detail is used, since we only need the original mesh in order to apply simplification operation on.

Although discrete level of detail seems bullet proof, due to its simplicity and efficiency, it has one major problem which occurs at the instance we switch from one mesh to another. A quick switch from one mesh to another creates a very distracting, and usually disturbing popping effect. This is logical since meshes won't exactly have the same boundaries, and generally, their edges differ extremely since this was the main reason they were created for, which is being very different in number of polygons.

There are several ways to reduce this effect of this disturbing switch between two meshes. Note that the word “reduce” was used because most if these ways do not totally remove the popping effect while switching.

7.1 Late switching

Our main problem with mesh switching is that the difference between the two meshes in question will be noticed by players, thus creating a very distracting and disturbing effect. Therefore “late switching” tries to resolve this problem from its root with no fancy solutions, which is by just switching late enough when the object if very distant from the active camera or very small on the screen, or in other words when the difference between the current and next mesh is barely noticeable. This sounds logical enough, because if the difference between the two meshes is barely noticeable, players won't be distracted by this level of detail switching.

This solution tries to solve our problem in a very direct and naive way, and the main problem with it is that it negates the main goal or advantage of using a level of detail engine. Remember that the main goal of using a level of detail engine is to reduce the number of polygons in a 3D scene as soon as possible in order to reduce rendering time, while trying not to affect the scene visually. Therefore, if we wait long enough in order to minimize the perceived difference between the two meshes, we will be rendering the high-polygon-mesh for a very long time before switching to a lower detailed version. It's also worth mentioning that the main problem may still exist, because we are still "hard switching" between the two meshes, without any kind of blending effect between them.

As a conclusion, late switching is considered a very naive and inefficient way to solve the popping effect, and it's not a practical solution for a general purpose application, although it could be of some use for a very narrow range of applications.

7.2 Hysteresis

This method tries to solve the popping effect from a different approach. It still uses the previous hard switching between two meshes, which is not implementing any kind of visual or geometric blending between the two, but it kind of offers a solution for the extreme pooping effect which occurs when an object is floating around the distance or size threshold, depending on which method is implemented. What it does it that it introduces a time lag into the LOD transitions, which means not switching to a different mesh as soon as the object passes the distance or size threshold, but instead, it waits for the object to get out of a certain interval before actually doing the switch.

To translate that description into computer or mathematical language, we are replacing the threshold value by a threshold interval. Let's assume we are using the distance criteria in order to determine which LOD to use, and a certain threshold distance is 100. If we

were using the usual hard switching, we would switch to a high detailed mesh if the object's distance form the active camera is less then 100, and switch to a low detailed version if that distance gets greater than 100. As you can probably imagine, this object will face fast periodic mesh level of detail switching if it is floating, or rotating around this threshold distance which is 100. Therefore, in order to reduce that effect, we are going to create an interval around the threshold limit. Let's take this interval to be [90,110], which means the a distance of 20 centered at our limit (100). Now if our object's distance from the viewer is less than 90 and increasing, we don't switch to a next low detailed version as soon as this distance becomes greater than 100 as we would usually do, but we wait till it passes the upper limit of the interval which is 110. On the other hand, if that distance is greater than 110 and decreasing, we will wait till it becomes less than 90 before we switch to the next high detailed mesh version. This solves our rapid LOD switching because if the object passes from one level of detail to the other, it will have to go back a significant amount of distance or size in order to switch back to its previous version. Now, an object floating around the threshold distance of 100 will no longer flip between two meshes.

This leads to a minor problem with using “hysteresis” to reduce the popping effect, which is we won't know ahead of time which mesh will be used at a certain distance or size. Using the previous example, if the object distance from the camera is 100, it could be using one of the two meshes, the one assigned for distances less than 100, or the one assigned for distances greater than 100. It depends on the path it entered the hysteresis interval through. If the distance was greater than 110, it will still be using the mesh intended for distances greater than 100, on the other hand, if the distance was less than 90, it will still be using the mesh intended for distances less than 100. This minor randomness becomes clear if two objects using the same model are approximately at the same distance from the active camera, and each one entered the hysteresis interval from a different side. As explained previously, these two objects will be using two different meshes although they are at the same distance from the viewer, which could be really annoying.

Making the hysteresis interval smaller can reduce the occurrences of this problem, because objects won't have to pass a greater distance in order to switch to a different LOD, but this brings us back to our main problem, which is switching between two meshes too often. So to conclude, there is no perfect hysteresis interval size. The greater it is, the less are the occurrences of extreme popping but objects may use different meshes depending on their camera-relative path, and the less this interval is, the more are the occurrences of the disturbing popping effects, but objects using two different meshes while at the same distance from the viewer will occur much less.

Astheimer and **Pöcke**'s experimentations concluded that a hysteresis of 10% gives optimum results, and that this percentage around the threshold limits is a good average to use, which doesn't favor any advantage over the other.

7.3 Alpha Blend

Alpha blending is the most used way to hide switching between two meshes, because of its simplicity and visual efficiency. It basically replaces the previous hard switching, which is directly replacing one mesh by another one during one frame, by drawing both meshes simultaneously using a different alpha value for each one, while changing these alpha values over time. In other words, we have expanded the switching time from one frame to a certain constant or non-constant depending on the used method. There are several way to implement alpha blending in order to hide the distracting popping effect, but they all revolve around the main idea of drawing the current and next meshes simultaneously while manipulating their alpha blend values.

7.3.1 Both meshes simultaneously

This method of using alpha blending in order to hide the annoying popping effect due to a mesh switch affects the alpha blend value of both meshes simultaneously. Let's take the same example we used for the hysteresis explanation. An object has a threshold distance of 100, which means that it must use a certain mesh if the distance that separates from the active camera is less than 100, and it must switch to a different mesh if that distance becomes greater than 100. We will to create an interval similar to the one we used for the hysteresis part, but this interval will be used as a fading region. What this means is that if the object is somewhere inside this interval, or fading region, two meshes will be rendered simultaneously but with different alpha blend values. We will use the same interval of 20 centered at 100, which is [90,110]. If the object's distance from the camera is less than 90, then we only draw the respective mesh in opaque mode, and if that distance is greater than 110, then we draw the other respective mesh also in opaque mode. Now to the middle part. If that distance is somewhere inside this interval, we will have to compute two alpha blend values, one for each mesh since we will be rendering both meshes. At first, the alpha blend value of the current mesh will be 1, or in other words the current used mesh will still be rendered in opaque mode, and the alpha blend value of the next mesh will be 0. Gradually, these alpha values will change until the one of the current mesh reaches 0, and the one of the next mesh reaches 1. At this point we will strop rendering both meshes, and start to use only the latter mesh whose alpha blend values changed from 0 to 1.

There are two approaches to handle the change for both alpha blend values, a distance/size based method, and a time based method.

7.3.1.1 Distance/Size based method

The distance method is going to be explained here, but it is completely similar to the size method. The only difference is that the threshold limits in the distance method represent the distance that separates the object from the active camera in the world coordinates system, while the threshold limits in the size methods represent the number of pixels occupied by the object in the screen coordinates system.

Using the previous interval [90,110], we said that at distance 90, the alpha blend value "λ" of the current mesh should be 1, while the alpha blend value of the second mesh should be "1 – λ" which is 0. As that distance varies inside that interval, so will λ, which will directly affect the alpha blend values of both meshes. We will have to come up with an equation that relates the current distance between the viewer and the object to the value of λ. A linear interpolation is more than sufficient for this calculation, therefore we will have to use the 2D line Cartesian equation to compute that linear relation.

Let's assume we have two points: A(90, 1) and B(110, 0). Having these two points, it will be easy to compute the Cartesian equation of this line, whose input represents the distance between the active camera and the object in question, and its output represents our λ value.

After computing this line equation (please note that a different Cartesian equation should be calculated for each threshold limit), we will be able to determine the alpha blend value λ for the current mesh, which is the direct output of this equation, while the alpha blend value for the next mesh will simply be "1 – λ".

- Current mesh: λ
- Next mesh: 1- λ

Alpha blending both meshes simultaneously has some disadvantages, like rendering two meshes instead of only one through the fading region, which increases the number of rendered polygons while the object is inside that region. This problem is common to all

alpha blending solutions, and its importance increases drastically if the fading region is relatively large, which will force our level of detail engine to render two meshes for a long time before it switches back to rendering one mesh only.. One of the other unique problems to this method is that while both objects are transparent, objects behind them may become apparent, which creates a visual artifact way worse then the popping effect.

7.3.1.2 Time based method

The time based method tries to solve one of the previously mentioned disadvantages of the distance/size methods. What it does is that instead of relating our alpha blend value λ to some uncontrolled factor like distance or size, we relate it to time. Imagine if the object stops somewhere inside the fading region for a long time. This could easily happen if we are dealing with a static object and the player stopped moving the camera at the point where that object was switching from one mesh to another. Should we render two meshes during that whole and maybe very long period?

Relating λ to time ensure that the transition from one mesh to another occurs over a constant pre-set amount of time. As soon as the object enters the fading region, start decreasing the alpha blend value λ at a constant rate until it reaches 0. Using the same technique as before, λ will be used as the alpha blend value of the current mesh, since it will vary from 1 to 0, while '$1 - \lambda$' will represent the alpha blend value of the next mesh.

Using the time based method can be used to solve the problem we faced with rendering two meshes for a relatively long time, but it still doesn't address the problem of seeing through both objects while they are both being rendered in transparent mode.

7.3.2 New mesh only

We faced a major problem with the previous method, which was seeing through the object while it was being transitioned from one mesh to another. This occurred because throughout the fading region, we were rendering both meshes in transparent mode using different alpha blending values. What this method suggests is that we only render the new mesh in transparent mode, while rendering the current mesh in opaque mode during all the time required for the object to exit the fading region.

Similarly to the previous method (where we rendered both the current mesh and the next mesh in transparent mode simultaneously), we will have to compute the Cartesian equation of a 2D line which represents the linear interpolation of the alpha blending value λ. After finding this equation, we will use the current camera-object distance or screen size (depending on which level of detail criteria you are using) as input, and the output will be the alpha blend value λ. Note that in the previous method, λ represented the blending value of the current LOD mesh, and $1 - \lambda$ represented the blending value of the next LOD mesh. Since using this method the current mesh is going to be rendered in opaque mode all the time, we will use λ to determine the alpha blending value of the next mesh, which is determined by the equation $1 - \lambda$. The next mesh will appear to fade in because its alpha blend value will start with 0 and increment to 1 as the object moves to the end of the fading region. When it reaches that point, the alpha blending value of the next mesh would have reached 1 which means it is being rendered in opaque mode as well, therefore, the current mesh can be removed, and we can continue rendering the next mesh (which obviously have become the current and only mesh) alone.

Drawing the current mesh in opaque mode will ensure that no objects behind the object in LOD transition will appear through it. Several z-buffer actions must be executed while rendering each mesh in order to avoid some visual artifacts. While rendering the current level of detail mesh, both testing with the z-buffer (z-test) and writing onto the z-buffer (z-write) should be enabled as if we are rendering a normal mesh. But when the time comes to render the next mesh, or in other words the mesh that is being faded in from completely transparent to completely opaque, we must turn off "z-write" to prevent "z-

fighting". "z-fighting" occurs when both LOD meshes are similar in object space but very different in image space, which occurs when both meshes have very different shading, texturing...

Two main advantages of using this method over the previous one exists. The first advantage is that objects won't appear through objects in level of detail transition, due to the fact that we are drawing an opaque object during the transition time. This issue was faced in the previous method where both meshes were rendered in transparent mode simultaneously, which created very disturbing and distracting effect. The second advantage is the decrease of rendering time. Rendering an object in transparent mode takes more time than rendering it in opaque mode, since for each rendered pixel, it will have to fetch the current buffer color and blend it with its current color, which obviously takes more time than just copying a certain color into the frame buffer. Therefore, this method is a bit faster than the previous one since only one object is being rendered in the relatively expensive transparent mode instead of two.

On the other hand, using this method has some disadvantages. Remember that an object in transition mode between two meshes will have its current mesh rendered in opaque mode during the fading region and then totally removed when the next mesh's alpha blending value reaches 1. This quick removal of the current mesh can create a distracting popping effect. Although it will be much less perceived compared to directly switching one mesh with another in one frame, it will however be noticed if the sizes of the two meshes in question are relatively different. This issue didn't exist in the previous method since we faded in the next mesh while fading out the current mesh, which created a smooth transition between the two meshes.

7.3.3 Both meshes separately

Each of the previous two method had some major disadvantages that makes any programmer think twice before implementing them. For the first method, which required that both the current mesh and the next mesh to be rendered in transparent mode simultaneously, which faded out the current mesh while fading in the next mesh, it had a very distracting and annoying of making the object transparent during the fading region, which allowed objects behind it, which obviously should not be visible in any way, to appear. As for the second method, which fixed the seeing through problem of the first method, it still did not totally fix the popping effect, because the current mesh was rendered in opaque mode during the fading region, and when the next mesh, which was faded in slowly, reached the opaque level, the current mesh was removed in one frame without any fade out effect.

What this method does is that it combines the advantages of both methods, which are fading in/out the current and next mesh, while not allowing objects behind the object in level of detail transition to be visible. This done by rendering both meshes but unlike the first method, we do that separately. Upon entering the fading region, we start fading in the next mesh until its alpha blending value reaches, or in other until it becomes completely opaque. What that happens, we start fading out the current mesh, until its alpha blending value becomes zero, or in other words the mesh becomes completely transparent. This ensures that at any point in time, at least one mesh will be rendered in opaque mode which will ensure that no objects behind the one in level of detail transition are being seen by the viewer. This method resolves also the popping effect of the second method (where only the next mesh was faded in and then the current mesh was removed without any fade out effect), because all meshes that are being added to the scene due to a level of detail transition are being faded in using alpha blending, and meshes removed from the scene are on the other hand being faded out also using alpha blending. No sudden adding or removing is being done using this method.

Like we did in the previous methods, some z-buffer functionalities must be manipulated during the rendering of each mesh. First of all, we should render the opaque mesh always

before rendering the transparent mesh. This means that during the first half of the fading region, we render the current mesh first since its being drawn in opaque mode, then we render the current mesh which is being drawn in transparent mode. Similarly, during the second half of the fading region, we render the next mesh first and then we render the current mesh. This is a very crucial rule: always render the opaque mesh first to ensure that no object behind the object in level of detail transition are being seen through it. The second important rule is that writing onto the z-buffer or "z-write" must be disabled while rendering the transparent mesh, and enabled while rendering the opaque mesh to prevent "z-fighting". As mentioned before, "z-fighting" occurs when both meshes are similar in object space but very different in image space. Of course testing with the z-buffer or "z-test" should always be enabled for both the opaque and the transparent mesh.

The same Cartesian equation of a 2D line must be generated in order to determine the current alpha blending value λ of each mesh. Depending on which level of detail criteria you are using (It could be distance or size as explained previously), you will use that criteria as an input for that function, and the output will be the alpha blending value λ. Let's talk about he first half of the fading region first. As we said, the current mesh will be rendered in opaque mode during this first half so it won't need the equation we generated for now. As for the next mesh, we usually determined its alpha blending value by computing "$1 - \lambda$", but since we want to reach the fully opaque rendering level within the first part of the fading region, or in other words twice as fast, we simply have to multiply that value, which leads to "$2(1 - \lambda)$". This means that when λ reaches 0.5, where usually the mesh should be rendered in half transparent mode since "$1 - \lambda$" would generate 0.5, the mesh will be drawn in full opaque mode because $2 \times 0.5 = 1$. Note that during all that time, the first mesh was being drawn ion opaque mode and before the transparent mesh. Now the meshes switch their roles, and the current mesh should start to fade out as the objects starts to exit the fading region. During the second part or half of that region, the next mesh will always be rendered in opaque mode and before the current mesh. In order to determine the alpha blending value of that mesh, we will use the same logic we used when alpha blending the next mesh, which is using the double of the value

generated by the Cartesian equation. Previously, the output of that equation which is λ was used as the alpha blending value of the current mesh. But since we have to fade out our mesh within half the fading region, we will have to double this value. Therefore, instead of using just λ as the alpha blending value, we will use "2.λ".

All this time doubling creates one minor disadvantage compared to previous methods though. As explained previously, the next mesh will reach its full opaque mode in the middle of the fading region, which leaves the other half for the currently used mesh to go from completely opaque to completely transparent. This means that the fading in of the next mesh is done twice as fast as it was done in previous methods. The same fact applies to fading out the current mesh because each mesh has only its respective half part of the fading region to fade itself within. This can be solved by doubling the range of the fading region, which will bring back the fading time of each mesh to its original amount, but this will lead to another problem which is extending the overall fading in/out time. Remembering that rendering in transparent mode is more expensive than rendering in opaque mode, since for reach rendered pixel, it will have to fetch the color from the frame buffer and blend it with its current color, which is obviously more expensive than overriding the frame buffer's current color. Therefore, doubling the time of rendering in transparent mode means doubling the computations, which is what we are trying to reduce by using level of detail.

7.4 Alpha LOD

Sometimes developers don't have enough time during a game's development process to create several mesh for each model in order to implement any of the previously mentioned level of detail methods. Creating several meshes for each model means more needed development time which is virtually never available, and more development time means more money to pay for employees, which is obviously one of the last things

development want. All these reasons combined led to the creation of a simple and very basic level of detail method called “Alpha LOD”.

Unlike previous level of detail methods where at least two meshes were required per model, alpha LOD requires only one mesh per model. Similarly to how object are rendered without a level of detail engine, the same model is used at any distance or size. One distance or size interval (depending on which level of detail criteria you are using) is needed though. This interval represents the fading region where we are going to fade out the object if it is getting away from the active camera, or to fade it in if it is getting closer to it. Let's assume that we are using distance as our level of detail criteria. For a particular we object, we set the lower bound of the interval to 600 and the upper bound to 800. What this means is that for any distance between the object in question and the viewer that is less than 600, we render the object normally, or in other words in opaque mode. On the other hand, for any distance greater than 800, we simply do not render that object. As you may have concluded on your own, we have to fade out the object and that is the purpose of creating the fading region or interval. A simpler version exists where we would only use one threshold value instead of an interval, where we would simply omit the object from the rendering pipeline after its distance from the active camera becomes greater than the pre-define threshold value. Of course, doing this will create a major popping effect, where object would just disappear or appear in front of players which is obviously a very annoying effect. So, as we did with previous methods, we will have to generate the Cartesian equation of a 2D line which represents the linear interpolation of the alpha blending value λ. The fading region gives us two point of that line which is enough to determine its Cartesian equation. The first point is (600,1) since at that distance, we want the alpha blending value λ to be 1 in order to render the object in opaque mode, and the second point is (800,0), sine at that distance we want to the object to become completely transparent before removing it from the rendering pipeline.

After determining this equation, we will use the current viewer-object distance as its input, and the result or output will represent the needed alpha blending value λ. When

that value becomes zero or less, we simply omit this object from the rendering pipeline, which reduces the scene's overall number of polygons. One of the main advantages of using “alpha LOD” is the need of only one mesh per model, which saves a lot of production time and money, and reduces the required amount of storage space. One other important advantage over some of the previous methods is that no z-buffer states are being manipulated during the fading out/in of any object. Remember that changing and device state is a bit expensive since several processes must be removed or added to the 3D pipeline when for each changed value.

This method is simple to implement and efficient, but it has its flaws. One of these flaws is that we are actually doing more calculation until the object's alpha blending value reaches zero. Remember that before entering the fading region, the object was being rendering in normal or opaque mode, but upon entering it, we started rendering it in transparent mode which is more expensive than rendering it in opaque mode, because for reach rendered pixel, it will have to fetch the color from the frame buffer and blend it with the current color, which is more expensive than just overriding the frame buffer's current color value. Another flaw of using alpha LOD is the inability to use it for very important objects in a game. In some games, the position of some objects is crucial to the gameplay, and their visibility is completely independent from the distance that separates them for the active camera. What this means is that using alpha LOD with these objects is impossible. Finally, we can safely say that alpha LOD is an application dependent method, since its use depends highly on the importance of the objects it is being applied on.

7.5 Geomorph

All the methods described before for reducing the popping effect during a discrete level of detail switch relied on blending the two meshes in image space. What this means is

that the level of detail engine comes into action only when it is time to render the object in question. What “geomorph” suggests is to handle that transition way earlier than the final image space, and to start blending the current and next mesh in the initial object space.

To morph a mesh into another one means to slowly transform the first mesh into the second one. What this means is that each vertex of the first mesh must be interpolated in order to reach a pre-chosen vertex from the second mesh during a set amount of time. And that does not apply to the geometric characteristics of the mesh, or in other words its vertices, but also to its texture maps and material etc...How is this helpful to for our discrete level of detail engine? Well since the main goal is reduce the distracting and even disturbing popping effect generated by the switch from one mesh to another less or more detailed one, we will use the geomoprhing technique in order to transform the current mesh into the next one, which obviously won't create the popping effect since no immediate changes are being done.

Several tasks must be done at pre-processing time in order to facilitate the morphing from one mesh into another. For each two consecutive level of detail mesh, we must “connect the correspondent vertices”. What this means is that each vertex from the high polygon mesh must identify with a vertex from the lower polygon mesh. In other words, each vertex must know the final destination it is heading towards during the morphing effect. An array of “destinations” should be saved for each two consecutive meshes. Now since we have both the source and destination of each vertex (The source being the vertex itself and the destination being the vertex of the chosen vertex from the next mesh), all we have to do is to add to the source this vector multiplied by a certain value. Note that this value must belong to the [0,1] interval. If this value is 0, then the final point or vertex is the same as the source. On the other hand, if that value is 1, the final point is the vertex of the nest mesh or the destination vertex.

At run time, depending on which level of detail criteria you are using (distance or size), you will use the previously called fading region as the region where all the morphing will occur. The distance criteria will be used for this example, but the same concept applies for the size criteria. Let's assume the interval where the mesh transition is occurring is [90,110]. As usual, we will have to generate the Cartesian equation of the 2D line which represents the interpolation of the value λ, which will be multiplied by the direction vector and then added to the source point in order to get the current position of the correspondent vertex. We know that at distance 90, the value of the variable λ should be 0 because at that distance, the position of each vertex is unchanged yet, and that at distance 110, the value of the variable λ should be 1, because each vertex should have reached its destination point. These two points which are (90,0) and (110,1) are enough to generate the Cartesian equation.

After determining this equation, we will use the current distance that separates the object from the active camera as input, and the output will represent the variable λ which will will multiply by the direction vector of each vertex, and then add it to the source vertex in order to determine its current position.

One of the main advantages of using morphing is that the transition looks much smoother than previous methods. Since blending is being done in object space and not in image space, the only popping effect that could occur is when the fading region is relatively small. What that's the case, you could see the vertices of the mesh in level of detail transition jump from one position to another. This is logical since they are being interpolated from one position to another in a very short amount of time. Another advantage is that none of the “z-buffer” states are being manipulated. Remember that changing any device state is a bit expensive, and changing it too often in one game loop will surely have a negative effect of the application's frame rate.

On the other hand, morphing a mesh onto another one has some disadvantages. One of these is the actual cost of interpolating all the vertices of the mesh in question. What if a

mesh had more than 15,000 vertices? Surely multiplying the interpolation variable λ by 15,000 direction vectors then adding them to the sources vertices is expensive, not to mention the relatively large amount of memory required to store these 15,000 direction vectors.

8 Non geometric LOD

So far, we have covered only one part, a big part that is, of the level of detail topic. All the previous LOD methods belong to the geometric part a level of detail engine, since their only goal is to reduce the overall number of polygons of a mesh, by simply replacing it by another simplified mesh of the same model, or by simplifying the main mesh using local simplification operators. This level of detail part is the most direct and logical way to reduce computation time since each time we remove a face or vertex, we are directly reducing th number of shading,texturing and any other vertex related functions that are applied to each vertex in the scene.

The other part of LOD is the non geometric reduction. When using geometric LOD, the generated less detailed mesh is intended to fill the same amount of screen space as the previous more detailed version, therefore, pixel fill-rate is not affected, and this is where non- geometric LOD comes into action. What it does is that it tries to reduce all these mentioned per vertex calculations without reducing the overall number of vertices. Remember that the main goal of reducing the number of vertices in a 3D scene was to reduce all the vertex related computations that came with them. But why not remove some of these tasks without removing the vertex itself? There are several level of detail non geometric methods to reduce the overall all time needed to render a 3D scene, which range from totally removing some per vertex computations to projecting the mesh on a 2D texture.

8.1 Shader LOD

As mentioned earlier, using geometric level of detail tried to reduce the number of vertices in order to reduce all the shader calculations that comes with each vertex in the scene. While rendering a certain mesh, each face has a certain number of texture maps that are referenced by it. When the time comes to render that face, the color of each pixel will be affected by each texture map. Some texture map are used as simple texture mapping, and others are used for some special effects like bump mapping, normal mapping, gloss mapping etc... As you can probably conclude, the per pixel cost is linearly dependent to the number of texture map being associated with that pixel. It's also worth mentioning that switching the device's state is considered a bit expensive, and therefore should minimized as much as possible.

What shader level of detail suggests is to reduce additional passes and features as the object becomes more distant. Let's take normal mapping as an example. Normal mapping is a way to simulate or give give the illusion of a high detail mesh while using a simplified mesh. This is done through shading using normals other that the vertices' normals. These normals are saved on a separate texture map called normal map, and normals are fetched from it during shading computation. As you can imagine, it is way more expensing than using flat shading, which is using the same normal for all the points of the same face. So to get back to our subject, as the object becomes more distant, we can start reducing the shader details assigned with it, like reducing features like bump mapping to simple flat shading. At extreme distances, the color of a face could be just reduced to a simple color fill without any special effects.

8.2 Effect scaling

Many games use special effects in order to increase the realism of the scene. Effects like casting shadows, particle systems are practically indispensable in modern games, and are used intensely in cutting edge games. What effect scaling suggests is to reduce the complexity of these effects as they get more and ore distant from the active camera, or in other words as these objects get smaller on the screen.

8.2.1 Shadow

Let's take the shadow casting for example. The most complex and realistic way to simulate it is to determine the object's silhouette from the light's perspective and project it onto nearby objects, or simply onto the ground. This method simulates realistically the nature's real shadow casting way, since by determining the silhouette of the object from the light's perspective, we are determining which area is blocking the light from passing through, which generates the respective shadow. As you can imagine, this method for generating the shadow shape is extremely expensive, since it requires some complex calculation in order to determine the silhouette of the object, not to mention that it needs to be calculated each game loop since that silhouette is highly dependent on the position of both the object and the light.

Using effect scaling of shadows means to reduce the computation time required to generates the shadow shape, which obviously means decreasing the shadow's precision. One of the methods to reduce this computation time is by using a pre-generated shadow shape and project it onto nearby objects or ground, using the light's position only to determine the projection direction. This saves a lot of computation time since we are not actually determining the object's real silhouette. This can be simplified even more by simply using a circle or ellipse as the shadow for any object, which still gives a great shadow illusion at great distances, while being very simple and inexpensive. As a conclusion, using effect scaling on shadow generation uses the real expensive method only when objects are near the viewing camera, while replacing these methods by simple fast methods when objects are far or in other words small in screen space.

8.2.2 Particle System

Particles systems are highly used in games to represent graphics that is hard to pre-model. In other words, artists can use some modeling programs like Maya or 3DS MAX to model meshes for static or articulated characters and objects, but it's hard to model a smoke, fire or water effect. These types of special effects are highly game dependent. For example, a smoke effect depends a lot on the size of the respective explosion, its direction and so on. Therefore, since it is illogical to model a smoke effect for all possible situations, particles systems are used in order to simulate these effects in real time.

Particle systems can benefit from effect scaling as well. Before going further into an example, it should be noted that particle systems are a huge graphic related area on their own, and not all particle system can benefit from effect scaling in the same way as others.

Let's take a smoke simulation:

A basic smoke effect simulation can be done by manipulating billboards (Billboards are 2D images which are always facing the viewer). When that smoke effect is initiated, a certain number of billboards are created, where the color, position and rotation of each one depend on the current life time of each particle. The more billboards that simulation has, the more flexible and appealing the smoke effect will be. Lets assume that 1000 billboards were used to represent the smoke effect. At close range, each billboard counts, but what if the same smoke effect is seen from a relatively great distance? Most of the billboards will overlap, therefore it's a huge waste of computation and rendering time to continue updating and rendering them all!

So what's the solution? Simply reduce the total number of billboards! At a great distance, 100 bigger billboards will create the same smoking simulation, but will require less updating, computation and rendering time. This implementation could be done the similarly to discrete LOD' one. For each distance interval, set the required number of billboards. At run time, when the distance separating the smoke effect from the viewer passes the pre-set threshold, switch to total number of billboards to the previous or next number accordingly.

8.3 Vertex processing LOD

As you know, each vertex passes through the entire 3D pipeline. From world/view/projection transformation, to shading, shader effects, rasterizing... It's quite clear that the time needed to produce 1 frame is directly affected by the number of processed vertices.

Almost all level of detail techniques try to minimize the total number of processed vertices, but we still have the same per vertex computation time. What this means is that although the total number of vertices was reduced by switching our model to a less detailed model, each of the remaining vertices is still going through the process as the one of the high detailed model. In other words, the per-vertex cost remained the same.

8.3.1 Shading

One of the most complex per-vertex process is shading. This process is linearly dependent on the number of available lights. What this method suggests is to reduce the total number of lights affecting the object as it moves farther from the viewer. To be fair to the removed light, we must add its ambient light to the object's average ambient. That

way , the removed light is still somehow affecting the final vertex' shaded color. As the gets smaller and smaller on the screen, it will only be lit by ambient lights, which means that skipping all the shading process.

8.4 Transformation LOD

Sometimes while animating a model, some vertices positions are altered in order to prevent unappealing effects. On of these disturbing effects occurs at joints, when the angle between the parent node and its children node becomes relatively large. Just imagine a human animation, where the angle between the arm and the forearm (The elbow joint) becomes large. If skinning is not used, a void will be seen between these two body parts, which is obviously unrealistic and unwanted. To prevent this problem from occurring, skinning is introduced, which prevents these unappealing effects.

As you know, while animation a model, each vertex will have to go through a different number of transformations depending on the related bones. In other words, each bone affecting a certain part from the model will contribute to that part's vertices' final transformation. The most common scenario nowadays is using weighted bones. What this means is that important bones will have a greater influence on the vertex' final transformation, which is obviously very logical.

There are two main methods to reduce the computation time of this process. The first method suggests ignoring these weighting information. This is done by reducing the number of bones affecting the vertices as the distance separating the animated model from the camera increases. Although this methid succeeds in reducing the overall computation time (less updated bones), it fails visually, because bones will be activated or disactivated instantly, which will create a disturbing popping effect (Similar to the popping effect created by the basic discrete LOD mentioned previously. This problem is

removed in the seconds method, which suggests reducing the bone weight gradually instead of just removing them. As the animated model gets father from the viewpoint, all bones will have their weight reduced except the most influential bone. The lowest detail form of this LOD technique will be ignoring all the bones except the most influential bone for each part.

8.5 Imposters

This level of detail technique suggests replacing our 3D geometric model by an image based LOD. What this means is that as the object gets farther from the view point, it would be replaced by a 2D imposter after passing a certain threshold. This imposter takes the form of a billboard (a 2D image which always faces the viewer). This greatly decreases computation time, since many steps are being omitted for that object, like vertex transformation for example. Doing so doesn't force us to leave the 2D imposter unshaded. As a matter of fact, there are many ways to shade a 2D imposter.

One shading method suggests simply applying a flat shading on the whole quad. Remember that we are replacing the 3D object by the 2D quad only when the former one becomes really far from the viewer, thus really small on the screen, therefore flat shading will be somehow acceptable. Another method suggests using a simple normal map approximating a sphere. Obviously, the spherical normals will be way different than the original normals, which mean that the final rendered shaded quad will be way different than shading the real 3D, but it will look better than flat shading, especially if the lights are moving.

8.6 Pre-rendered texture imposters

On relatively slow machines (Like PDAs, phones, watches etc...), implementing a 3D pipeline is a bit out of reach. Therefore, instead of using 3D meshes (which is impossible), artists still draw 3D models, and take snapshots of that model from different angles. At run time, as the object rotates, we switch from on snapshot to another, which will give the illusion of a 3D model. This is the same way movies work! You have a series of still frames, but when scene one after the other at a high rate, you will see the "illusion" of motion. Obviously, the more snapshots we have of a certain 3D model, the smoother its rotation will look.

The biggest challenges this method faces is dealing with point of view changes. We can't know ahead of time which angle will our object be seen from, and obviously, we can't create different snapshots for all possible point of views! A solution to this problem is to prepare the rotation snapshots of the 3D objects around the X, Y & Z axis, and at runtime, using the angle between the viewer and the object forward axis, you can determine axis (X, Y or Z) is closest to the calculated axis, and use its corresponding snapshot.

8.7 Render to texture

Similarly to the imposter method, this method suggests rendering our 3D model to an off screen surface or texture when it passes a certain distance threshold. In other words, when the distance separating the object from the view point becomes greater than a pre-set limit, we stop treating that object a 3D entity, and we render it to a surface. As long as that distance is still greater than the threshold, we keep rendering that same surface over an over again. This will drastically reduce the computation time of that object, since it is no longer being processed in the 3D pipeline. This method uses less storage than the previous one since sprites are created at runtime, compared to saving several sprites. In other words words, no memory is devoted to "stand by" sprites.

This method has some limitations though. It is wroth reminding the readers that rendering to an off screen texture, which is heavily used in this method, is slower than rendering directly to the frame buffer. Therefore we will get a real optimization when the same surface is rendered over several frames. Another main limitation is the infamous popping effect. When switching from the between 2 2D surfaces (this occurs when the distance between the object and the camera becomes passes the 2nd level threshold), a disturbing popping effect might occur because the previously rendered texture would most probably be way different than the current one, depending on the object's orientation. This disturbing popping effect could be reduced by smoothly blending between the current surface and the next one (Very similar to the way we switch between two discrete LOD meshes).

8.8 Geometric imposters

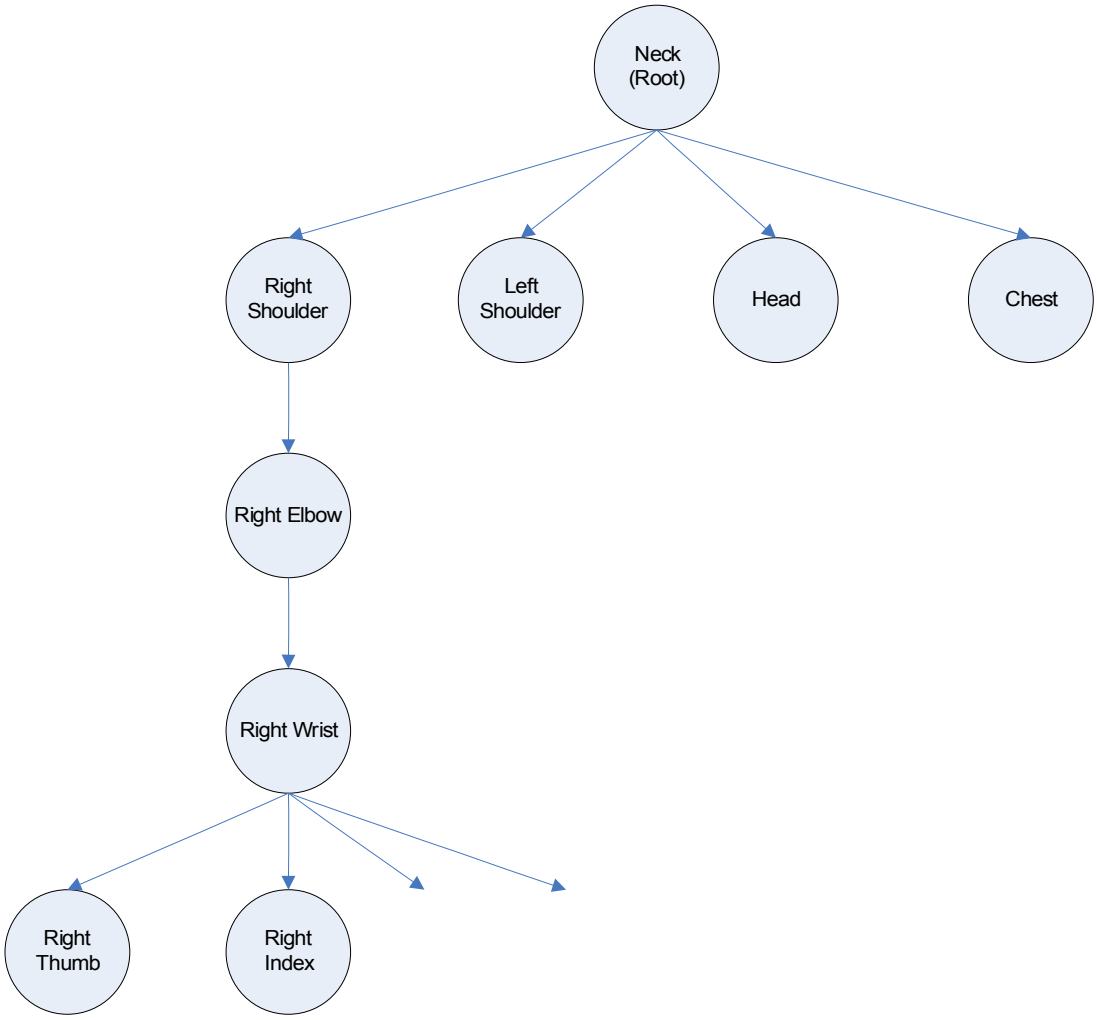
This method is mainly used for articulated meshes. At great distances, fine motion details won't be visible, mainly because all the mesh's part will overlap on the screen. So why updating each bone of that mesh? Geometric imposters suggests replacing an articulated mesh by a rigid one, which will greatly reduce the computation time needed by each bone to update its matrices. This method might seem a bit as a brute force solution, therefore other methods could be used to avoid directly replacing an articulated mesh by a rigid one. One of these methods suggests removing bones transformations one by one as the object gets farther from the camera. This way, we will have a smoother transformation from the fully articulated mesh to the rigid one. Removing skinning at great distances is also applicable, since fine skinning won't be noticed anyway when the object's screen size is relatively very small.

The main advantages of using geometric imposters is the preservation of a realistic silhouette regardless of the viewing angle, and the preservation of the object's lighting information. These advantages (compared to using the previously mentioned 2D imposters) exist solely because we are still using a 3D mesh! But obviously, we will have to have several pre generated 3D meshes which we will use at run time to replace the articulated parts of our animated mesh, which would greatly increase the required storage memory for each mesh.

9 Motion LOD

9.1 Hierarchy structure

Articulated models are widely used in applications and games in general. Unlike static models, this articulation allows for a more realistic character representation, since it allows multiple static shapes to be attached among each other using joints.
An articulated model is saved as a tree in memory. The root of the tree can have any number of children bones, which in turn can have any number of children bones of their own, and so on.



Each bone or joint represents a unique set of three dimensional transformations called key frames, along with a respective time stamp for each key frame. These key frames are provided by the artist, and they are usually built using a modeling package. At run time, the application should, for each bone, generate a certain transformation matrix by interpolating between the transformation of 2 key frames. These 2 key frames are determined by the current application time, which will be converted to animation specific time in order to determine which key frames to choose.

General transformation matrices are usually built using 3 transformations: Scale, Rotation and Translation.

A scale value is saved as 3 float numbers: The first one scales the shape on the X axis, the second one scales it on the Y axis and the last one of the Z axis.
A rotation value is saved as a quaternion, which is a structure of 4 float numbers. It represents the axis of rotation along with the angle of rotation.
A translation value is saved as a 3D vector, which is a structure if 3 float number. The first one determines the object's translation on the X axis, the second one determines its translation on the Y axis, and the last one determines its translation of the Z axis.

In the update part of the game (each game loop), the transformation matrix of the root bone will be computed by concatenating its translation, rotation and scaling matrices. Then this matrix will be passed as a "parent matrix" to each of its children bones, and will be concatenated with each of these bones' own transformation matrices. Again, each bone will pass its world transformation matrix (Which is the result of the concatenation of their own transformation matrix with their parent's one) to all of its children and so on. As you can see, any change in the transformation of any bone will be propagated to all the bones located beneath that bone in the animation tree, which creates the transformation relativity of the articulated model.

9.2 Introduction

Similar to discrete LOD, there exists a level of detail reduction method for articulated models. An articulated model is a group of static models all connected to each other in a hierarchy system. The articulated model is built using bones, which are nodes in the previously mentioned hierarchy structure, each holding several transformations saved as key frames. These key frames are provided by the artist, and it's the programmer's responsibility to import the articulated hierarchy along with the key frames of each, and to interpolate between the key frames at run time. At any given time, the current time of the animation will be used in order to determine the current transformation of each bone.

Then, each bone's transformation will be applied to the group of polygons associated or linked to it.

9.3 Static LOD on articulated models

The same static LOD concept which was applied to non-articulated models can be applied to articulated models too. As mentioned previously, each bone of the articulated model will generate a certain transformation at run-time. This transformation will be applied to the group of polygons associated or linked with this bone. Applying static LOD to an articulated model could be done by replacing each group of polygons by a simpler groups (fewer count) as the object's level of detail decreases. These groups of polygons are provided by the artist, and choosing which one to display depends on many factors like size of screen, distance from view point...(Refer to the discrete LOD section from more information).

9.4 Continuous level of detail for articulated models

In addition to simply replacing each bone's group of vertices in the articulated object by a simpler group (lower polygon count), which is almost identical to discrete level of detail, we can also apply continuous level of detail separately on each mesh, which reduces the popping effect resulting from switching between one mesh and another one. This method doesn't require any additional models from the artist. The high detail version will be simplified at run time when the object's level of detail is decreased. (Refer to the continuous LOD section for more details).

9.5 Articulated model specific LOD

But these level of detail techniques are not specific to articulated models, because they target each part of the articulated model as a separate entity, just as we do with static models. In other words, they optimize the articulated model the same way they optimize any static model, which is good generally, but optimizing an articulated model can be enhanced even more, by targeting another “expensive point” in it, which is the transformation interpolations.

Articulated objects can also be simplified in order to alleviate their computation time. An articulated model is a group of bones, where each bone is connected to another one, inheriting its position, orientation, scale and any other type of transformation. In other words, if bone A is the parent of bone B, then all B's data (Position, Rotation etc...) are in A's coordinates system, and all A's translations, rotations, scaling and other transformation will affect bone B. Note that there is a technique called “skinning”, where the positions of the vertices are altered, each according to its position from the joint, and to the weights of the corresponding bones. This means that each vertex is affected by the transformations of one or more bones.

9.6 Hierarchy replacement

9.6.1 Description

A more specific approach suggests replacing the entire articulated object by another one which contains less bones, thus less transformation interpolations each frame. Although the new mesh contains less bones, it must at least preserve the general shape of the object. An indispensable method to preserve the shape of the object is to make sure that the end effectors (The last point in the articulated body. An articulated model can have more than one end effector) aren't affected much in the low detail version of the

articulated mesh. This means that their positions in each key frame of the low detail articulated model should be as close as possible (if not exactly the same) to their corresponding positions in the high detail articulated model. Remember that the lower detail version of the model will only be used when the object is far from the view point, and its final screen size is relatively very small.

Another important condition to make the less detailed articulated model look as much as possible the higher one is to have the same key frames. Key frames are unique distinct frames provided by the artist, each with its own transformation and time. At run-time, the game, or application in general, will interpolated from one key frame to the next one, which means interpolating the transformations of the key frames, taking into consideration the time of each one.

Similarly to the discrete level of detail of static models, the artist is responsible for providing the different version of the same model, each with a different level of detail, which means different number of bones, thus different transformations which will eventually lead to less and less number of transformation interpolations at run-time.

9.6.2 Drawback

The drawback of the method is that it requires much more storage memory, because for each animated model, the artist will have to provide several versions each with its own level of detail. This will also affect the amount of needed memory at runtime, because all the animations models will have to be loaded at the same time, since switching from one to the other will be dependent on many run-time factors, like distance from camera, on screen size... In other words, every LOD version of the same articulated model will have to be “ready” at any moment, because the object's level of detail can dramatically vary from high to low or vice versa during gameplay.

9.6.3 Optimization

One of these disadvantages could be avoided by implementing a software based level of detail reduction solution. What this means is that the artist is now responsible for providing the most detailed version of the articulated model, and while loading this model, we generate several lower detail versions of it. While this technique solves the "storage memory" problem, due to the fact that all the low level detail versions are generated at run-time using a single version (the high detail one), it doesn't reduce in any way the amount of "run time" memory, because all model's versions, the original one and the generated ones, will have to be ready in case they're needed at run time. The drawback of generating the different level of detail models using an algorithm as opposed to being provided by the artist is that they won't be as polished obviously. In other words, when the artist provides all the versions, it is somehow guaranteed that all these versions are modeled in a way to resemble the original high detail version, while the generated ones, although they will resemble the high detail version, but their error terms will obviously be a bit greater than of the ones provided by the artist.

10 Priority based animation level of detail

As explained previously, each bone in the animation hierarchy will compute its own transformation matrix by interpolating between two transformations of two key frames. Interpolating between between two key frames is done by interpolating between:

- Two scale structures (Usually 2 3D vectors)
- Two rotation structures (Usually 2 quaternions)
- Two translation structures (Usually 2 3D vectors)

This interpolation is expensive computation wise, but is obviously needed in order to animate the articulated model. But do we really need to interpolate the transformation of *each* bone during *each* game loop? The number of times a bone's transformation is interpolated can be decreased as the model's final screen size becomes smaller, without affecting it visually. A naïve but simple method to achieve this computation reduction would be to reduce the number of transformation of all the bones simultaneously each certain number of game loops.
The problem with this method is that it treats all the bones of the animation hierarchy equally. In other words, all the bones will skip their transformation interpolation the same number of times, which is a bit “unfair”, because certain bones have a greater effect on the overall shape of the animation compared to other bones.

Several factors determine how much a certain bone contributes to the overall shape of the articulated model. Some of these factors are embedded in the transformation matrix of the bone itself, like the rotation, translation and scaling amounts, and other factors are determined by the “state” of the bone in the hierarchy tree, like its depth and its number of children bones.

In order to determine the degree of contribution of each bone to the overall shape of the articulated model, let's examine how each of these factors individually affects the transformation interpolation from one key frame to the next of each bone.

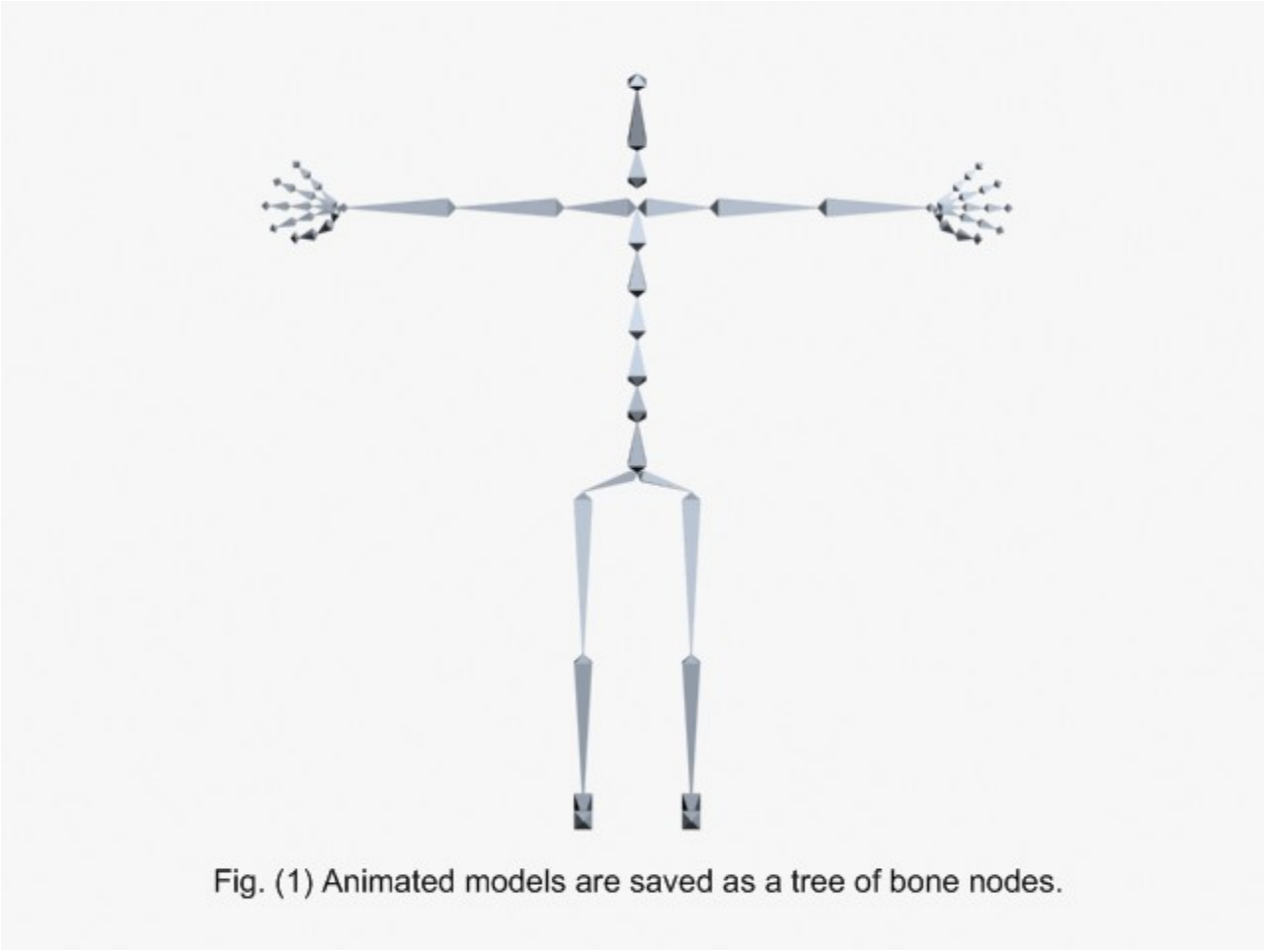


Fig. (1) Animated models are saved as a tree of bone nodes.

10.1 Bone rotation

Each bone has its own set of rotation per key frame, which is totally independent from other bones' rotations. If a bone has a rotation (R_A) of 90 degrees in 2 frames while another bone has a rotation (R_B) of 40 degrees in the same number of frames, will skipping the same number of interpolated frames for both rotations (R_A & R_B) have the same visual effect? Obviously, skipping a certain number of interpolation steps from rotation (R_B) won't be as noticeable as skipping the same number of interpolation for

rotation (R_A), due to the fact that the degree of transformation of rotation R_A is much greater then the one of rotation R_B.

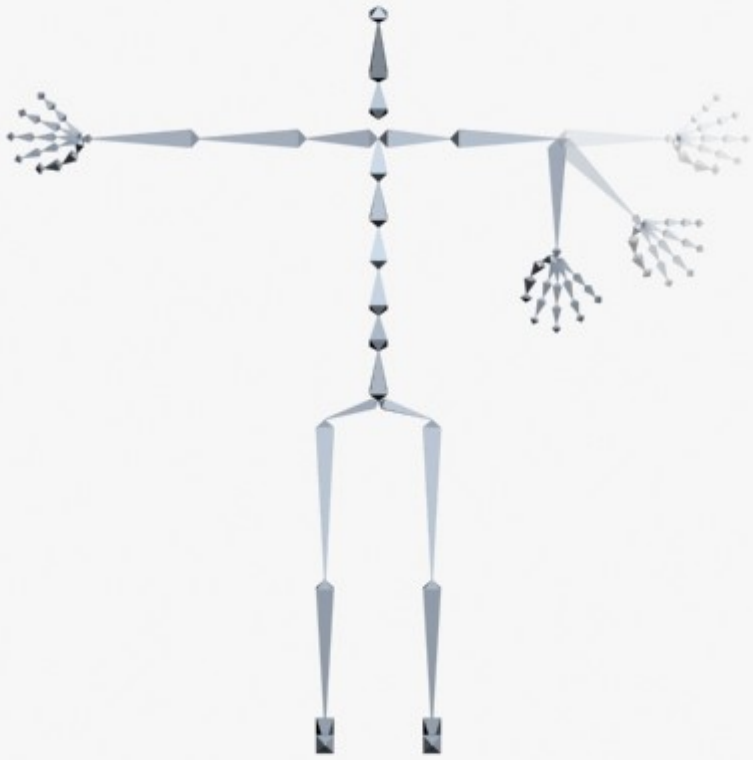


Fig. (2a) Bone rotation of 90 degrees during 2 frames.

In the above example, rotation R_A, which is a rotation of 90 degrees is applied to the bone for the duration of 2 frames.

The most translucent group of bones represents the state before any transformation is applied.

The semi-translucent group of bones represents the state of the bones after the first interpolation, which is at frame n+1.

The opaque group of bones represents the final state after the rotation R_A is applied, which is at frame n+2.

Obviously, skipping the in between interpolation (at frame n+1) of the rotation R_A in the above hierarchy animation will greatly affect the overall shape of the articulated model. The main reason is that the visual difference between the shape of the animation at frame n and frame n+2 is relatively great.

Let's apply rotation R_B to the same bone:

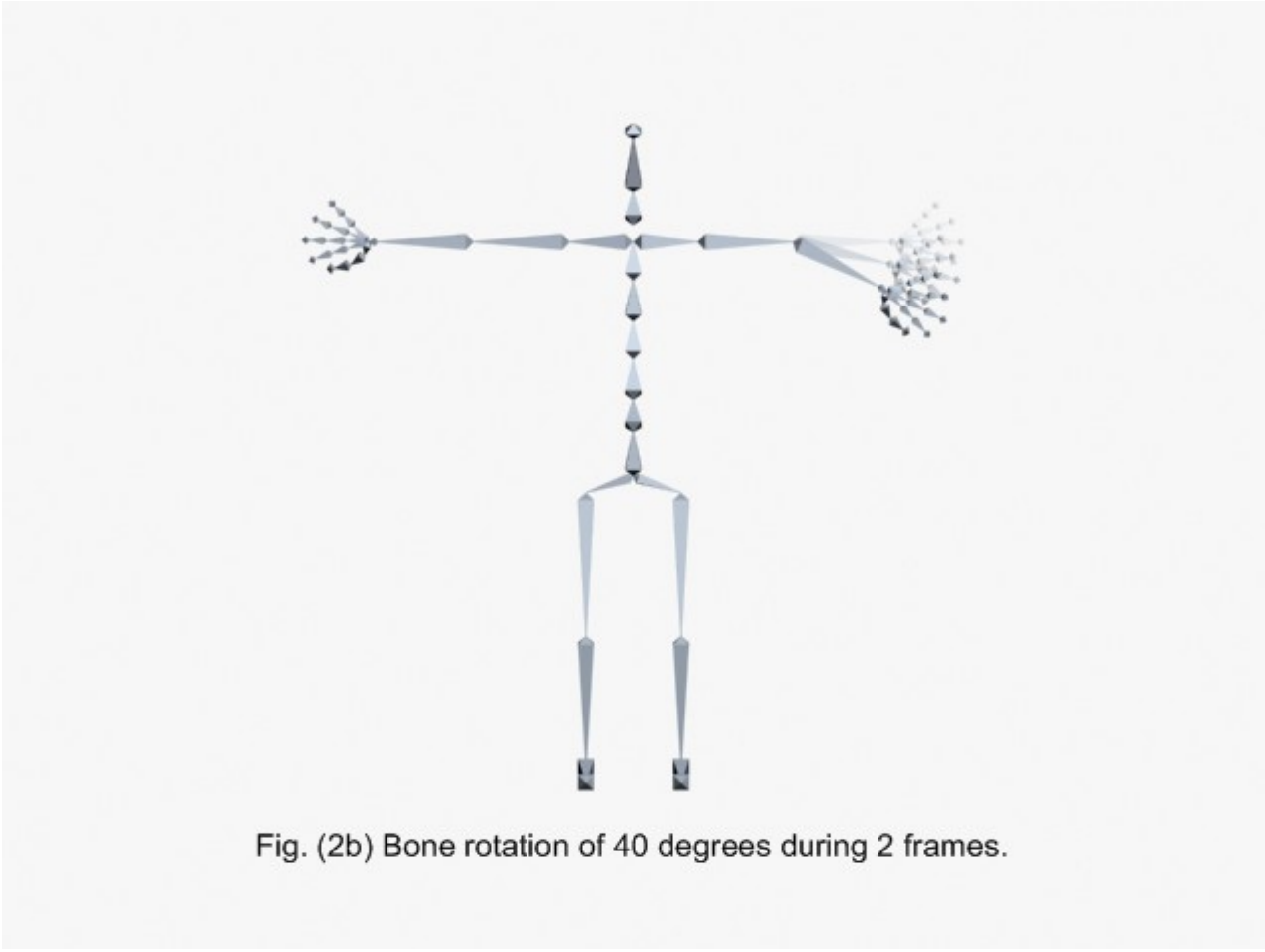


Fig. (2b) Bone rotation of 40 degrees during 2 frames.

In the above example, rotation R_B, which is a rotation of 40 degrees is applied to the bone for the duration of 2 frames.
The most translucent group of bones represents the state before any transformation is applied.

The semi-translucent group of bones represents the state of the bones after the first interpolation, which is at frame n+1.
The opaque group of bones represents the final state after the rotation R_B is applied, which is at frame n+2.
We can see that skipping the middle interpolation (at frame n+1) of rotation R_B will be less noticeable than skipping the middle interpolation of the rotation R_A in the previous example, due to the fact that the absolute difference between the shape of the animated model at frame n and frame n+2 is much less when rotation R_B is applied compared to R_A.

Our animation level of detail algorithm should take the rotation difference between 2 consecutive key frames into consideration when determining how much a certain bone contributes to the overall shape of the animated model.

10.2 Bone translation

In an animated model, each bone has its own set of translation vectors per key frame, which is totally independent from other bones' translation vectors. We should check the translation effect on the overall shape of the articulated model similarly to the way we checked the effect of rotating a bone has on it. If a bone is translated by a translation vector $\vec{T}_A(-20,0,0)$ in 2 frames while another bone is translated by a translation vector $\vec{T}_B(-5,0,0)$ during the same number of frames, will skipping the same number of interpolated frames have the same visual effect? The greater the translation vector is, the more it will be visible when interpolating the key frames. Therefore, skipping a certain number of interpolation steps for bone affected by the translation vector $\vec{T}_B(5,5,0)$ won't be as noticeable as skipping the same number of transformation interpolations from the bone affected by the translation vector $\vec{T}_A(-20,0,0)$, because

the degree of transformation of translation $\vec{T}_A$ is relatively greater than the one of translation $\vec{T}_B$.

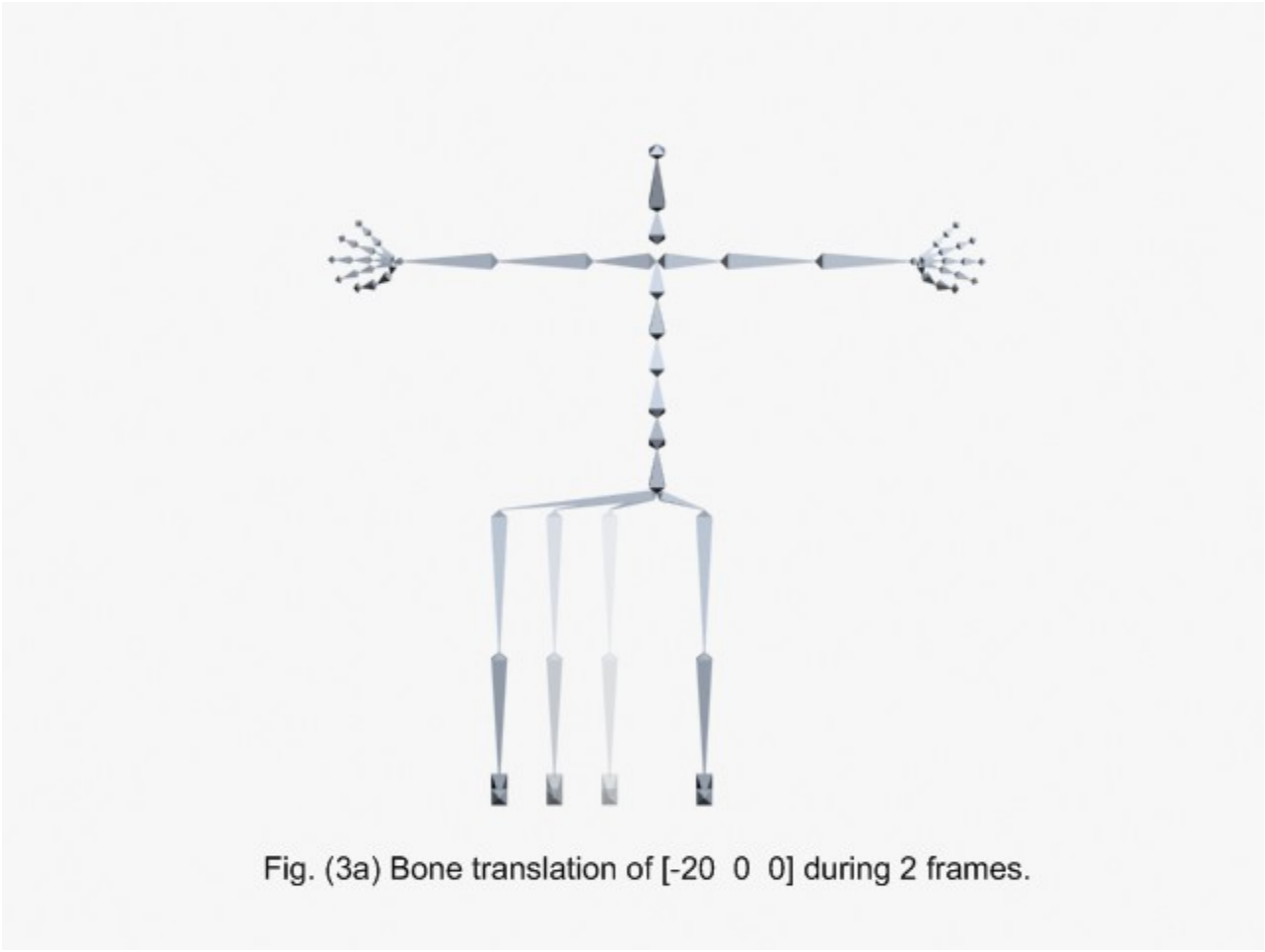


Fig. (3a) Bone translation of [-20 0 0] during 2 frames.

In the above example, $\vec{T}_A$, which is a translation of $(-20,0,0)$ is applied to the bone for the duration of 2 frames.

The most translucent group of bones represents the state before any transformation is applied.

The semi-translucent group of bones represents the state of the bones after the first interpolation, which is at frame n+1.

The opaque group of bones represents the final state after the translation $\vec{T}_A$ is applied, which is at frame n+2.

Obviously, skipping the in between interpolation (at frame n+1) of the translation $\vec{T}_A$ in the above hierarchy animation will greatly affect the overall shape of the articulated model. The main reason is that the visual difference between the shape of the animation at frame n and frame n+2 is relatively great.

Let's apply the translation $\vec{T}_B$ whose vector is $(-5, 0, 0)$ to the same bone:

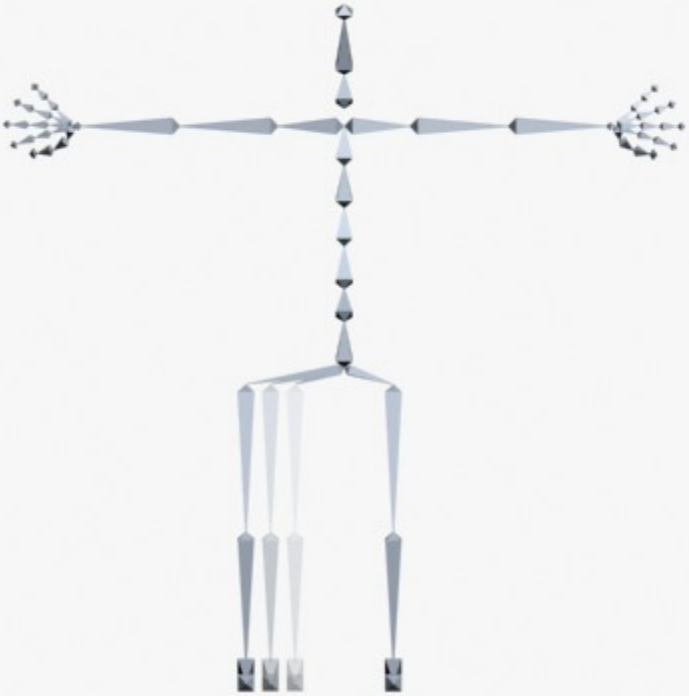


Fig. (3b) Bone translation of [-5 0 0] during 2 frames.

In the above example, translation $\vec{T}_B$, whose vector is $(-5, 0, 0)$ is applied to the bone for the duration of 2 frames.

The most translucent group of bones represents the state before any transformation is applied.

The semi-translucent group of bones represents the state of the bones after the first interpolation, which is at frame n+1.

The opaque group of bones represents the final state after the translation $\vec{T}_B$ is applied, which is at frame n+2.
We can see that skipping the middle interpolation (at frame n+1) of the translation $\vec{T}_B$ will be less noticeable visually, compared to skipping the middle interpolation of the translation $\vec{T}_A$ in the previous example, due to the fact that the visual difference between the shape of the animated model at frame n and frame n+2 is much less when translation $\vec{T}_B$ is applied compared to $\vec{T}_A$.

Finally, it's safe to say that the translation transformation difference between 2 consecutive key frames should be taken into consideration when determining the degree of contribution of each bone to the shape of the animated model.

10.3 Bone scaling

Each bone in an animated model has a set of scaling value values, each one for a separate key frame. This scaling value of each bone independent from other bones' scaling values. Since scaling a bone directly affects the offset of all the vertices associated with it relatively to the center, then the difference between the scaling values of 2 consecutive key frames will have a direct impact on the overall shape of the articulated model in case we decide to skip some transformation interpolation steps.
If a certain bone is to be transformed by the scaling transformation $\vec{S}_A(2.5, 2.5, 1)$ during 2 frames while another bone is to be transformed by a vector $\vec{S}_B(1.5, 1.5, 1)$ during the same number of frames, then skipping the same number of interpolated frames for both scaling transformations $\vec{S}_A$ and $\vec{S}_B$ will have a different visual effect on the animation model. The greater the scaling vector is, the more it will be visible when interpolating the key frames, and the more noticeable will be skipping one of its transformation interpolation steps.

Let's compare how the scaling transformations $\vec{S}_A(2.5, 2.5, 1)$ and $\vec{S}_B(1.5, 1.5, 1)$ separately affect the look of our articulated model.

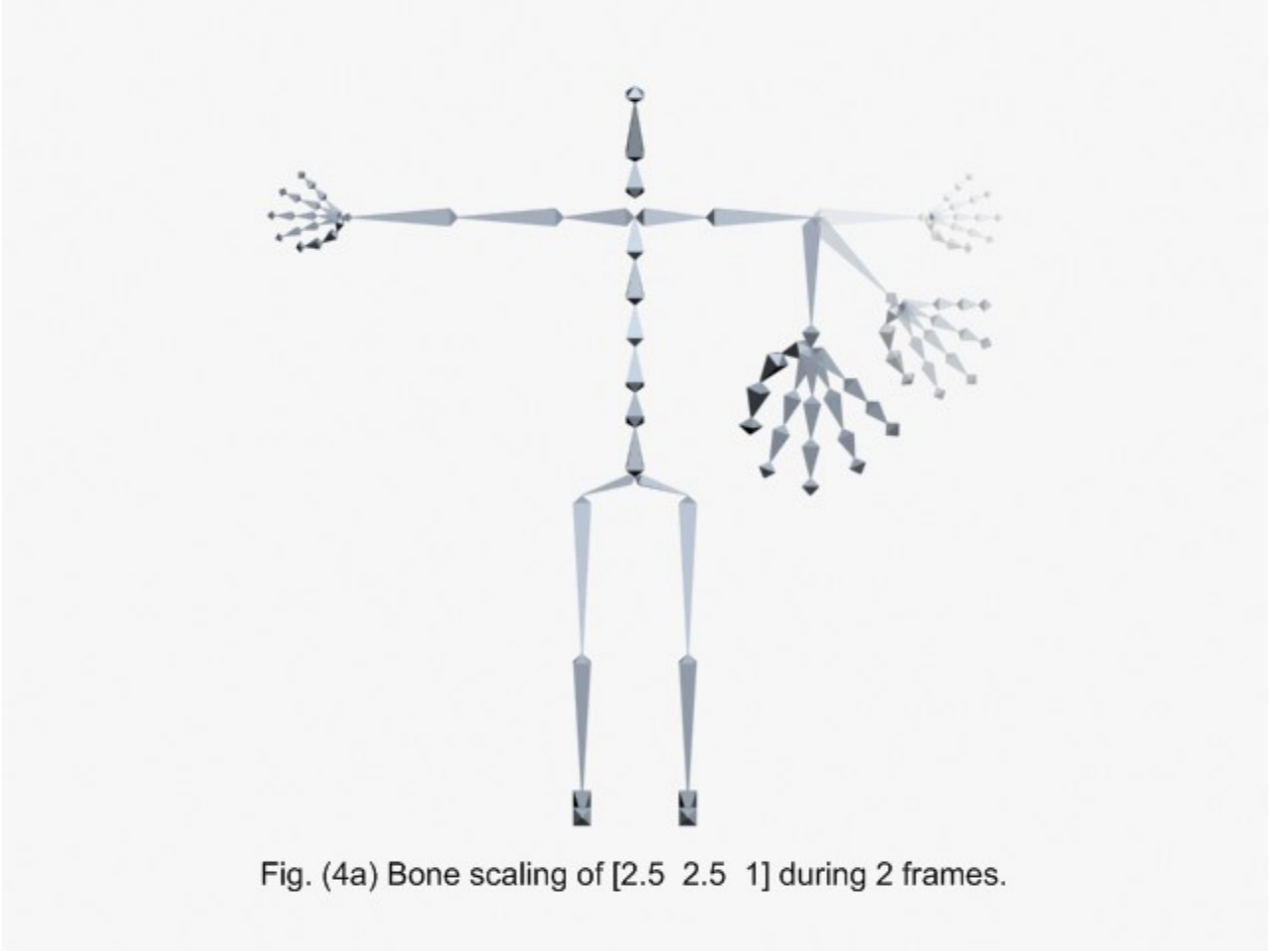


Fig. (4a) Bone scaling of [2.5 2.5 1] during 2 frames.

In the above example, $\vec{S}_A$, which is a scaling transformation of $(2.5, 2.5, 0)$ is applied to the bone for the duration of 2 frames.

The most translucent group of bones represents the state before any transformation is applied.

The semi-translucent group of bones represents the state of the bones after the first interpolation, which is at frame n+1.

The opaque group of bones represents the final state after the scaling transformation $\vec{S}_A$ is applied, which is at frame n+2.

Obviously, skipping the in between interpolation (at frame n+1) of the scaling transformation $\vec{S}_A$ in the above hierarchy animation will greatly affect the overall shape of the articulated model. This is due to the fact that the degree of transformation difference at frame n and frame n+2 is considered relatively high after being affected by the scaling transformation $\vec{S}_A$. Note that the "degree of transformation" means how much it is being affected visually.

Let's apply the scaling transformation $\vec{S}_B$ whose vector is $(1.5, 1.5, 0)$ to the same bone:

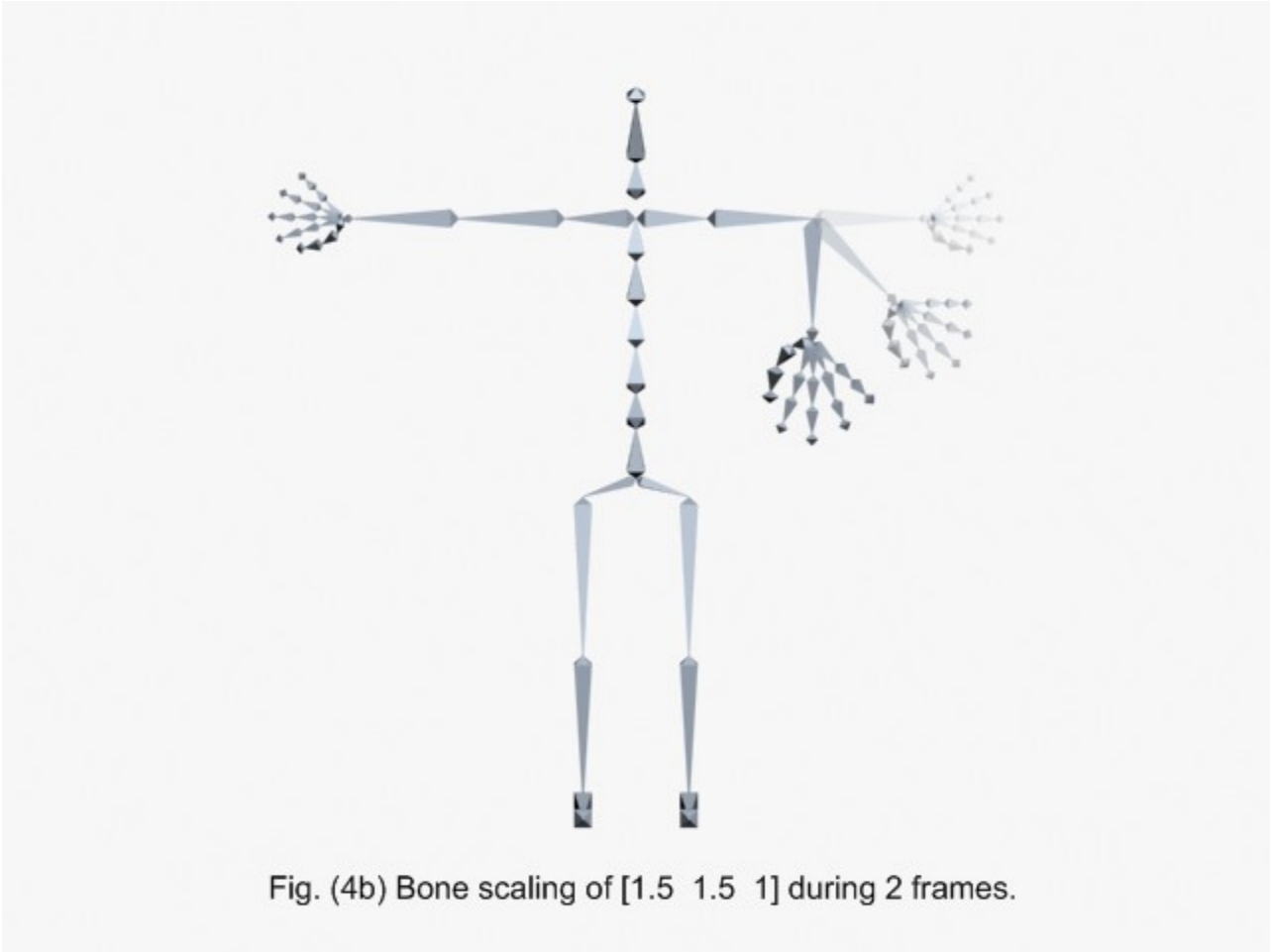


Fig. (4b) Bone scaling of [1.5 1.5 1] during 2 frames.

In the above example, the scaling transformation $\vec{S}_B$, whose vector is $(1.5, 1.5, 0)$ is applied to the bone for the duration of 2 frames.

The most translucent group of bones represents the state before any scaling is applied.
The semi-translucent group of bones represents the state of the bones after the first interpolation, which is at frame n+1.
The opaque group of bones represents the final state after the scaling $\vec{S}_B$ is applied, which is at frame n+2.
We can see that skipping the middle interpolation (at frame n+1) of the scaling transformation $\vec{S}_B$ will be less noticeable visually, compared to skipping the middle interpolation of the scaling transformation $\vec{S}_A$ in the previous example, mainly because the final size difference at frame n and frame n+2 is clearly greater when the scaling transformation $\vec{S}_A$ is applied compared to $\vec{S}_B$.

In conclusion, we can safely say that the sacling transformation difference between 2 consecutive key frames should be taken into consideration when determining the degree of contribution of each bone to the shape of the animated model.

10.4 Bone's number of children

In the articulated model description, we mentioned that each bone has a set of children bones. Each one of these children bones will concatenate its parent bone's transformation matrix with its own transformation matrix, in order to generate its final transformation matrix.
This final transformation matrix of the bone in question will be passed to its own children which will do the same process, until this propagation traverses all the bones of the articulated model (Remember that the articulated model is saved as a tree of bones).
Updating the animation model can be described as illustrated in the following recursive function:

Bone Update Function

$OwnMatrix = Interpolate\,Own\,Transformation$

$WorldMatrix = Concatenate\,ParentMatrix\,with\,OwnMatrix$

$\omega = Number\,of\;Children$

$For\;Each\,Children\,Bone\,B_i, i \in [1, \omega]$

$Bone\,Update\,Function(B_i)$

It is clear that the transformation matrix of any bone B_d (d being the depth of of the bone B), which is built by concatenating its translation, rotation and scaling transformations, will be propagated to all the bones located under the bone B_d $\{B_e : e > d, B_e\,Direct / Indirect\,child\,of\;B_d\}$ (Remember that the articulated character is saved as a tree of bones).

Because each bone's final transformation matrix needs its parent world transformation matrix, then skipping a transformation interpolation for any bone would affect the final world transformations of all the bones who are direct or indirect children of that bone. We can conclude that skipping a transformation matrix interpolation of a bone having many direct and indirect children bones will have a greater negative visual effect on the overall shape of the articulated character, compared to skipping a transformation interpolation of a bone having few direct or indirect children bones.

The "number of children" error term $\delta^{B}_{NumberOfChildren}$ of skipping a transformation interpolation is relative to the total number of direct and indirect children bones.

$$B^{\omega}$$

$$\omega = number\ of\ direct \quad \text{and} \quad children\ of\ bone\ B$$

$$\phi_{PerChild} = error\ factor\ per\ bone$$

$$\delta^{B^{\omega}}_{Number\ Of\ Children} = \omega \times \phi_{PerChild}$$

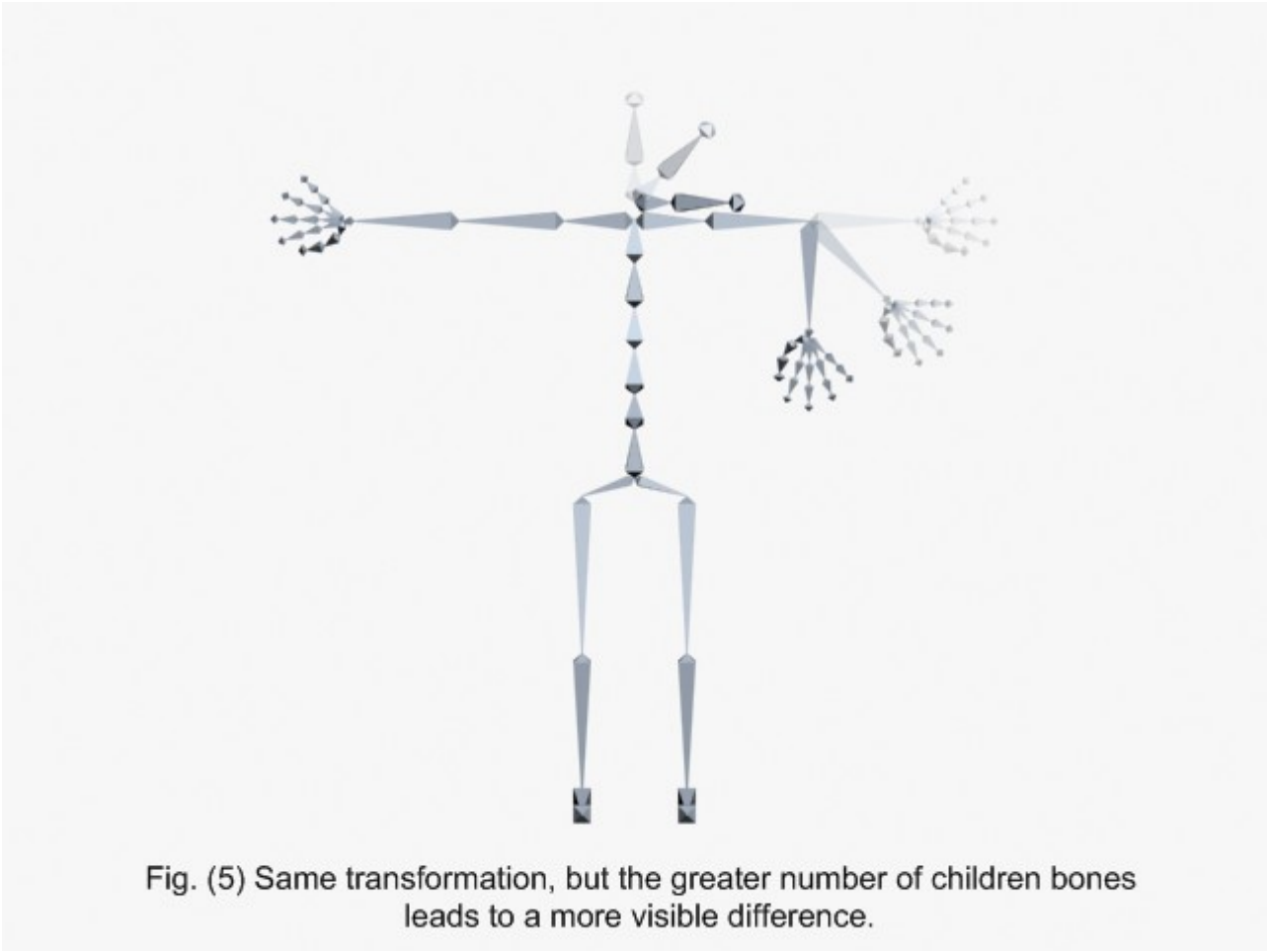


Fig. (5) Same transformation, but the greater number of children bones leads to a more visible difference.

10.5 Bone's depth in the animation hierarchy

The higher a bone is in the animation hierarchy, the greater its transformation matrix influences the rest of the model. As explained previously, the animation model is saved as a tree of bones, where each bones computes its final transformation matrix by concatenating its own local transformation matrix with the its parent world transformation matrix.

This means that any change in the transformation of any bone B_d (d being the depth of of the bone B) will have a direct impact on all the bones B_e

$\{B_e : e > d\ ,\ B_e\ Direct / Indirect\ child\ of\ B_d\}$ who are direct or indirect children of the bone B_d .

Having this in mind, it seems logical that skipping the a certain number of transformation interpolations of a bone B_2 located at the second depth level in the animation hierarchy will distort the final shape of the animation model much more than skipping the same number of transformation interpolations of a bone B_6 located at the sixth depth level in the animation hierarchy.

The "depth" error term δ^{B}_{Depth} of skipping a transformation interpolation of a certain is relative to the depth of the bone itself.

$$B_d$$

$$d = Depth\ of\ bone\ B$$

$$D = Maximum\ depth \in the\ animation$$

$$\phi_{Depth} = error\ factor\ per\ single\ depth\ difference$$

$$\delta_{Depth} = \phi_{Depth} \times \frac{D}{d}$$

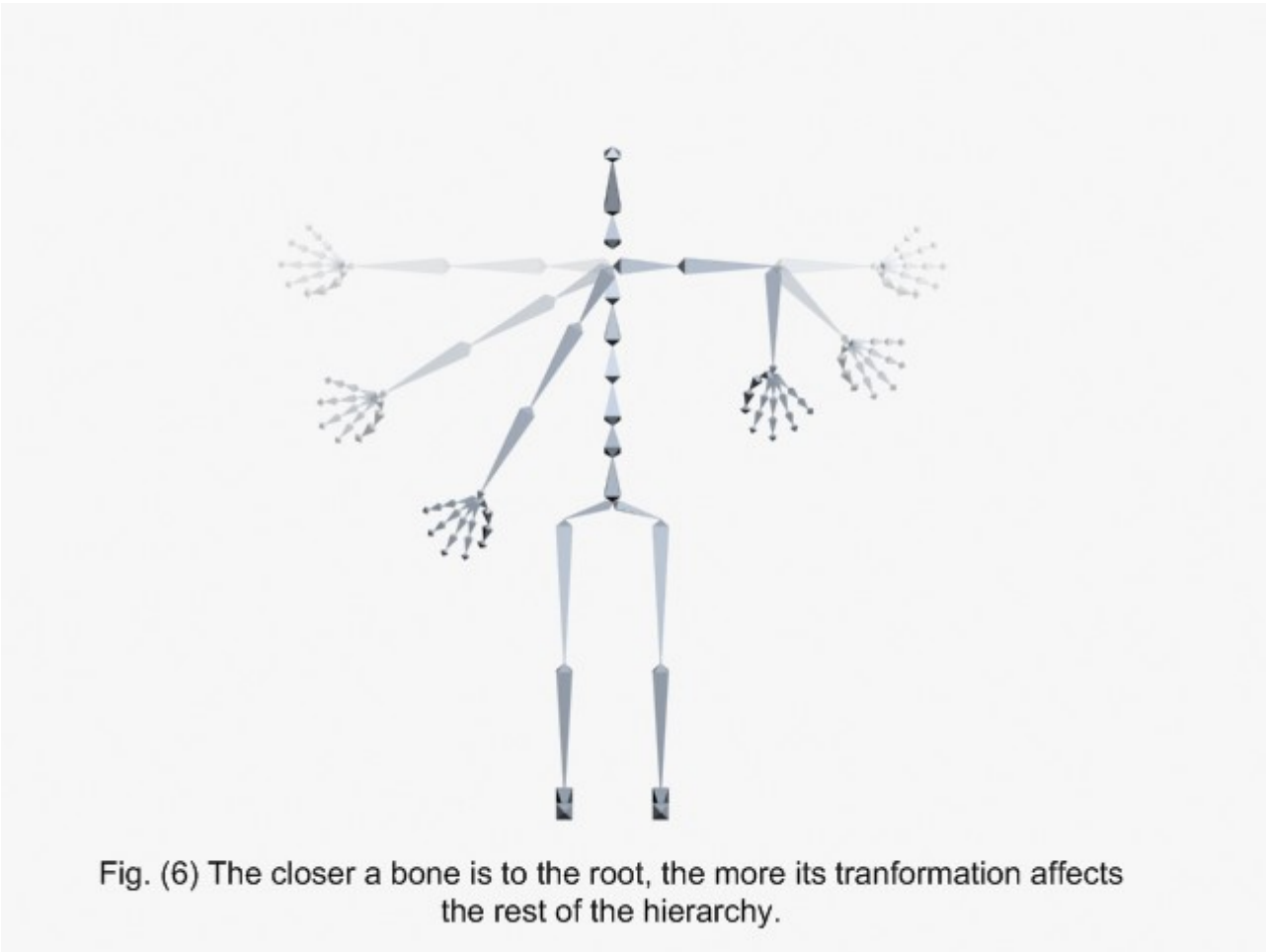


Fig. (6) The closer a bone is to the root, the more its tranformation affects the rest of the hierarchy.

10.6 Combining the factors

We can see that computing how much a bone contributes to the overall shape of the articulated model is done by checking how much skipping some of its transformation interpolations affects the animated model. The greater this effect is, the more the bone is considered "*important*".

Therefore, each bone should have a certain *priority* based on its contribution when producing the final shape of the animation.

This priority value will determine how much a bone will “skip” its transformation interpolations. The more a bone contributes to the final shape of the animation, the higher its priority will be, and skipping its transformation interpolations will be relatively less than other lower priority bones.
In other words, the higher the priority of a certain bone is, the less its “skipping transformation interpolation” frequency will be.
In the implementation description of the animation, we mentioned that that animation will be saved as a tree of bones, each having its key frames. Each key frame is independent, which means a bone's current transformation depends solely on the transformations of its previous and next key frames.
Having a single priority for each bone seems not enough, because a bone can be transformed greatly between certain key frames, while having a relatively minor transformation change between other key frames. Therefore, each key frame of each bone will have its own priority, whose value depends on how much its transformation is “different” from the previous key frame's transformation.

11 Priority

11.1 Translation priority

At each stage of the animation, a bone can translate towards or away from its parent bone. Usually artists don't manipulate the bones' translation vectors a lot when creating articulated models, they rely mostly on the bones' rotations transformations. Note that a bone's transformation (Translation, rotation, scale, ...) is relative to its parent bone. If a bone's position changes dramatically, it means that reducing its level of detail would highly affect the overall shape of the hierarchy, due to the high effect it has on it. This level of detail implementation will assign a *translation* priority to each bone relative to its

position changing ratio. The more a bone is translated to or away from its parent bone, the higher its translation priority will be.

11.2 Rotation Priority

The more a bone rotates around its parent, the more it contributes to the model's animation. Therefore, reducing the level of detail of a bone that rotates around its parent bone would greatly affect the shape of the object, compared to reducing the level of detail of a bone that is aligned with it parent, or at least doesn't rotate a lot around it. This level of detail technique will make sure that the more a bone rotates around its parent bone, the highest its *rotation* level of detail priority will be.

11.3 Scale Priority

Each bone has a scaling factor which will be applied to all the vertices locked to it. Furthermore, since scaling a bone will automatically affect the scaling transformations of all its children bones, which means affecting their vertices too. It's clear that the scale value of each bone play a very important role in the articulated hierarchy, and therefore should be treated as any other component. Since reducing the level of detail of a bone whose scale factor varies a lot will have a high impact on the overall shape of the articulated animation, then a bone with a greatly variable scale factor will have a greater *scale* level of detail priority.
This means that the more a bone is scaled up or down, the higher its scale priority will be, in order to reduce the number of time this bone skips its transformation interpolation.

11.4 Number of Children Priority

This priority factor targets the bones' number of children. In a hierarchical animation structure, each bone is connected to a single parent bone, but it can have any number of children bones of its own. While updating a certain animation, starting from the root, each bone will calculate its own local transformation, which will be passed to its children as “parent transformation”. Each children will concatenate its local transformation with the one provided by its parent, and will pass the result to its own children. This process is done throughout the entire animation structure. With this said, it is obvious that a bone's transformation will affect all of its children transformations, and of all the bones located beneath it in the bone tree. Therefore, while computing the bones' priorities at loading time, bones having few children bones will have smaller *number of children* priorities compared to bones with relatively more children bones.
In other words, bones with a relatively great number of indirect and direct children bones will have greater “*number of children*” priorities in order to minimize the number of times they skip their transformation interpolations.

11.5 Depth Priority

The closer a bone is to the root, the more reducing its level of detail will affect the rest of the hierarchy. On the other hand, if a bone is relatively far from the root, reducing its level of detail will logically have a smaller effect on the bone hierarchy, because its transformation is propagated to a smaller number of children bones. Note that this distance is bone based, which means it is a whole number that starts by 0 at the root, and is incremented by 1 for each bone depth. This level of detail algorithm will automatically assign a higher *depth* level of detail priority to bones closer to the root compared to bones relatively deep in the animation hierarchy.

11.6 User-Set priority

Finally, each bone will have a user-set priority. This priority is not generated by the algorithm and doesn't rely on any of the previously mentioned factors (translation, rotation, scale, number of children and depth). The user is allowed to explicitly set a priority for each bone, which will be averaged with the algorithmically generated priorities. If the user doesn't specify any priority, then the level of detail priority generating algorithm will assign the same *user-set* priority for all the bones.
The purpose of this priority is to give users some flexibility in determining the final priority value of each key frame of each bone.

In conclusion, the final priority value will be the average value of these 6 priorities.

12 Implementation

12.1 Gathering the animated model's level of detail information

Before assigning a priority for each key frame of each bone of the articulated character, we need to gather some information about the entire animation during loading time.

The priorities that need to be computed for each key frame of each bone are:

- Translation priority
- Rotation priority
- Scale priority
- Number of children priority
- Depth priority

The algorithm starts by looping through all the bones, and for each bone, it will loop through all its key frames in order to determine all the "maximum transformations". These maximum transformations are:

- The maximum translation difference between 2 consecutive frames of any bone.
- The maximum rotation difference between 2 consecutive frames of any bone.
- The maximum scale difference between 2 consecutive frames of any bone.
- The maximum number of direct and indirect children of any bone.
- The maximum depth value of the articulated model.

The animation will have 5 variables dedicated to saving the maximum computed value of each level of detail priority field.

$MaxTranslation = 0.0$
$MaxRotation = 0.0$
$MaxScale = 0.0$
$MaxNumberOfChildren = 0$
$MaxDepth = 0$

Starting from *the root :*

Check *if current number of children is greater than the maximum*
Check *if current depth is greater than the maximum*

For each children bone b_i
    For each key frame b_i^j
        Compute translation difference
        Check *if translation difference is greater than the current maximum*

        Compute rotation difference
        Check *if rotation difference is greater than the current maximum*

        Compute scale difference
        Check *if scale difference is greater than the current maximum*

12.2 Computing the translation difference

As mentioned previously, each key frame of each bone has its own translation vector. We will compute the length of this translation vector in order to determine how much each bone is moving relatively to its parent bone.

The greater the length of the translation vector, the more distant the bone is from parent. If the length of the translation vectors of the consecutive key frames is very different, then the bone is considered to be "agitated".

Let $\vec{T}_n$ be the translation vector of the bone during the current key frame, and $\vec{T}_{n+1}$ be the translation vector of the next key frame. $\vec{\Delta}_T$ is the difference between $\vec{T}_{n+1}$ and $\vec{T}_n$. Finally, l_T is the length of the vector $\vec{\Delta}_T$.

$$\vec{\Delta}_T = \vec{T}_{n+1} - \vec{T}_n$$
$$l_T = \left|\vec{\Delta}_T\right|$$

After computing the translation amount of the bone from key frame n to key frame n+1, we compare the length of that translation difference to the animation's current maximum translation difference, and we save it in case it was greater.

$$if\,(l_T > MaxTranslation)$$
$$MaxTranslation = l_T$$

12.3 Computing the rotation difference

Aside from having a translation vector, each key frame of each bone has its own rotation value, usually saved in the form of a quaternion. In order to compute the difference

between two rotations, we will compute the dot product of the two quaternions representing the rotation transformation of the key key frame n and n+1.
The greater this quaternion dot product is, the more the bone is rotating around its parent, and its rotation priority for this particular key frame should be relative to that quaternion dot product. If the dot product of the rotation quaternions of any 2 consecutive key frames, then again the bone is considered to be "agitated".
Let $\vec{R}_n$ be the rotation quaternion of the bone during the current key frame, and $\vec{R_{n+1}}$ be the rotation quaternion of the next key frame. β is the dot product result between $\vec{R}_n$ and $\vec{R_{n+1}}$.

$$\beta = \vec{R_{n+1}} \cdot \vec{R}_n$$

After computing the dot product of the 2 quaternions representing the rotations of 2 consecutive key frames, we compare the result β to the animation's current maximum rotation dot product, replacing it if it is greater.

$$if\,(\beta > MaxRotation)$$
$$MaxRotation = \beta$$

12.4 Computing the scale difference

The final type of bone transformations used besides the translation and rotation is the scale transformation. This transformation scales all the vertices which are associated with the bone in question. The scaling transformation is saved as a 3D vector.
The greater the length of the scale vector, the more the bone is being scaled up or down relatively to its parent. If the length of the scaling vectors of the consecutive key frames is very different, then the bone is considered to be very varying scale wise.

Let $\vec{S}_n$ be the translation vector of the bone during the current key frame, and $\vec{S_{n+1}}$ be the translation vector of the next key frame. $\vec{\Delta}_S$ is the difference between $\vec{S_{n+1}}$ and $\vec{S}_n$. Finally, l_S is the length of the vector $\vec{\Delta}_S$.

$$\vec{\Delta}_S = \vec{S_{n+1}} - \vec{S}_n$$
$$l_S = |\vec{\Delta}_S|$$

After computing the translation amount of the bone from key frame n to key frame n+1, we compare the length of that translation difference to the animation's current maximum translation difference, and we save it in case it was greater.

$$if\,(l_S > MaxScale)$$
$$MaxScale = l_S$$

12.5 Computing the maximum number of children bones per bone

Since this value is directly related to the bone hierarchy of the articulated model, we don't need to loop through all the key frames of each bone in order to determine it.

Note that the needed value is not only the maximum of direct children of any single bone, but the number of both direct and indirect children. It's obvious that the root bone will have the greatest number of direct and indirect children bones, but we will have to recursively compute each bone's number of direct children and pass the result back to the parent, which will add the received result to its own number of direct children bones and so on, until the root has collected all the results. This back propagation is needed in order to determine each bone's number of direct and indirect children bones.

While looping through all the ones in order to determine the maximum translation, rotation and scale transformations, We will ad the current bone's number of direct children bones to the total result.

$$Starting \text{ from } the\ root$$

$$Function\ F_{TotalChildren}: Compute\ number\ of\ children\ bones$$

$$MaxNumberOfChildren \ += \ Number\ Of\ Direct\ Children\ Bones$$

$$For\ each\ children\ bone\ b_i$$
$$F_{TotalChildren}(b_i)$$

12.6 Computing the maximum depth of the articulated hierarchy

Similarly to the maximum number of children bones per bone, the maximum depth of the hierarchy is not affected by the number of bones nor the number of key frames per bone.
An articulated hierarchy can have a total of 5 bones and having a maximum depth of 1. This occurs when 4 of these 5 bones are all the direct children of the root bone. On the other hand, an articulated hierarchy can have a total of 3 bones and having a maximum depth of 2. It all depends on how the bones are connected among each other.
While looping through all the ones in order to determine the maximum translation, rotation and scale transformations, we should keep track of the current depth, replacing the animations maximum depth value if needed.

$$if\ (CurrentDepth > MaxDepth)$$
$$MaxDepth = CurrentDepth$$

Now that we gathered all the required information about the articulated model, we can proceed to the next step, which is computing the per key frame priority of each bone.

12.7 Computing the bones' priorities

Each bone will have its own per key frame priority, which represents how much the key frame in question contributes to the overall shape of the articulated character. The more a bone is agitated during a certain key frame, the higher that latter's priority will be.
A higher priority value will reduce the number of times a bone skips its transformation interpolation. On the other hand, a relatively low priority will allow the bone to skip a greater number of transformation interpolations.
Note that this process should be done just one at loading time.

The priority of each key frame of each bone will be computed using the same criteria which was used while collecting the articulated model's information, which is:

- Translation
- Rotation
- Scaling
- Number of children
- Depth
- And finally, the user-set priority will be treated as any other factor, and will added when computing the final priority value.

A total of 6 variables, each associated with a different priority criteria will be needed in order to compute a key frame's level of detail priority.

$P_T : Translation\ Priority$
$P_R : Rotation\ Priority$
$P_S : Scale\ Priority$
$P_D : Depth\ Priority$
$P_{Num} : Number\ of\ Children$
$P_{User} : User-set\ priority$
$P_{b_i^j} : Priority\ of\ key\ frame\ j\ of\ bone\ i$

In order to compute the value of each sub priority, we will compute the ratio between the key frame's data and the maximum data that was collected in the previous step (Gathering the animated model's level of detail information). The bones which generate the highest criteria value will eventually end up with relatively high priorities, allowing them to minimize the number of times they skip their transformation interpolations.
Again, we will loop through all the bones and compare their data to the maximum values collected previously.

Starting from *the root :*

For each children bone b_i
For each key frame b_i^j

$$P_T = \vec{\Delta}_T^{b_i^j} \ / \ MaxTranslation$$
$$P_R = \beta^{b_i^j} \ / \ MaxRotation$$
$$P_S = \vec{\Delta}_S^{b_i^j} \ / \ MaxScale$$
$$P_{Num} = Number\ of\ Chilren \,/\, MaxNumberOfChildren$$
$$P_{Depth} = CurrentDepth / MaxDepth$$

$$P_{b_i}^j = (P_T + P_R + P_S + P_{Num} + P_{Depth} + P_{User}) \ / \ 6$$

12.8 Weighted average

After computing the value of the six sub priorities of each key frame of every bone in the articulated model, we computed the final priority of that key frame by getting the average of the aforementioned six sub priorities. In other words, the algorithm treated these sub priorities equally, by implicitly assigning the same *weight* for each one:

$$w = \frac{1}{6}$$

$$P_{b_i}^{j} = w.P_T + w.P_R + w.P_S + w.P_{Num} + w.P_{Depth} + w.P_{User}$$

But what if the user wants to emphasize on any particular sub priority? For example, if the key frames of the bones of the animated model are pre-known (The artist usually can provide such information) to be very different rotation wise, which means the difference between 2 consecutive rotations is generally very high compared to the translation or scaling difference, it would be more logical to make the rotation sub priority have a greater effect the final key frame's priority.

The weighted average of the 6 sub priorities allows users to have a finer level of flexibility (compared to the user-set sub priority factor) when determining the final sub priority of each key frame of the animated model's bones, by allowing them to choose a different weight for each sub priority, where the sum of all the weights is equal to 1.0

w_T: *Weight of the Translation* sub *priority*
w_R: *Weight of the Rotation* sub *priority*
w_S: *Weight of the Sranslation* sub *priority*
w_{Num}: *Weight of the number of children* sub *priority*
w_{Depth}: *Weight of the Depth* sub *priority*
w_{User}: *Weight of the User set* sub *priority*

$$P^{j}_{b_i} = w_T . P_T + w_R . P_R + w_S . P_S + w_{Num} . P_{Num} + w_{Depth} . P_{Depth} + w_{User} . P_{User}$$

As an extreme case, the user might decide to assign the entire weight, which is 1.0 to a single sub priority if needed. If this is the case, each key frame of every bone in the animated model will end up taking that factor's (rotation, translation, scale..) correspondent priority without averaging it with the other ones.

It's the user's responsibility to adjust the weights of each sub priority to allow the animation level of detail algorithm to adapt to different animations. For example, if the animation manipulates the scale factors of its bones more frequently and with greater differences than it manipulates their translation factors, then the weight associated with the scale sub priority should obviously be greater than the translation's one.

12.9 Using a bone's priority to manipulate its level of detail

The static priority of each key frame of each bone of the articulated model will be used to manipulate the level of detail of the respective bone. Although we used the entire hierarchy's information to compute all the static priorities, we will be manipulating the level of detail of each bone separately. In other words, the static priority of each key frame of each bone wasn't deduced out of the bone's information alone.

The first step was to loop though the entire tree of bones, and gather its information:

- MaxTranslation: Maximum translation during one key frame
- MaxRotation: Maximum rotation during one key frame
- MaxScale: Maximum scaling during one frame
- MaxNumberOfChildren: Maximum number of children bones per bone
- MaxDepth: Maximum depth level of the animated model

In step 2, the level of detail algorithm looped again through all the key frames of each bone of the animated model, and computed its own unique 5 sub priorities:

- P_T : Translation priority
- P_R : Rotation priority
- P_S : Scale Priority
- P_{Num} : Number of children priority
- P_{Depth} : Hierarchy depth priority
- P_{User} : User-set priority (This sub priority is set by the user)

Finally, the algorithm computed the average of these 6 sub priorities in order to determine the final static priority of each key frame of each bone.

The main purpose of any level of detail technique is to reduce the detail of the object of its detail can't be perceived anymore. This occurs when the object's final project size of the screen is relatively small compared to its original size. In a 3D scene, this occurs when the distance separating the object and the camera (viewer) increases. Therefore, this level of detail algorithm will use the distance separating the viewer and the animated model when manipulating the latter's level of detail.
Due to the fact that an object's location in the 3D scene can vary almost randomly in each game loop, this means that the final level of detail of any object will frame-based. In other words, the level of detail of an object in frame is totally independent from the previous and next frame's level of detail.

12.10 Applying Level of Detail using a Distance Range

In order to use the object's position when computing its current frame's level of detail, we need to specify a distance range: [MinDistance – MaxDistance]. This LOD range is relative the viewer. In other words, this distance is computed by calculating the distance seperating the object from the camera. A **dynamic ratio** Ω will be computed for each object, depending on the distance discussed previously.

- If the object's position is less than the lower bound of the distance range, then it is considered to be greatly visible, and every bit of its detail is perceived. Therefore, any object closer to the viewer than the range's lower bound won't have its level decreased, which means all of its bones will interpolate their transformations normally as they do if there wasn't any level of detail technique applied.

  $\Omega = 1.0$

- If the object's position is greater than the upper bound of the distance range, then it is considered to be barely visible, and all of its original detail is barely perceived, if even at all. Therefore, any object whose distance separating it from the viewer is greater than the upper bound of the LOD range will skip all of its transformation interpolations.

  $\Omega = 0.0$

- Finally, if the distance separating the object from the viewer is within the level of detail range, then the object dynamic ratio is computed as follows:

$$\Omega = (Current\ Distance - MinDistance)\ /\ (MaxDistance - MinDistance)$$

- This will make sure that the closer an object is to the viewer, the greater its dynamic ratio will be. On the other hand, its dynamic ratio will converge to 0 as it goes away from the viewer.

Now that when the per-frame dynamic ratio of each animated object is calculated, we can finally compute each bone's current level of detail by scaling its current frame's static priority by the object's dynamic ratio.

for each bone : i
Get current key frame : j
Get static ratio of key frame j : P_i^j
Scale static priority by dynamic ratio Ω
$CurrentLod = P_i^j \times \Omega$

Finally, each bone's current level of detail is calculated, based on its current key frame, its static priority and the object dynamic ratio, which will allow us to manipulate the number of times this bone's transformation will be interpolated.
Remember that in order to decrease the needed computation time when updating the animated model, some of its bones' transformation interpolations will be skipped, each according to the bones current static priority (based on current key frame) and the object dynamic ratio (based on object's distance from the viewer).

An articulated model is updated based on the time elapsed from the last time it was updated. Therefore, this level of detail algorithm will use the time factor in order to decide if the current transformation interpolation of a certain bone should be skipped. The final priority of the bone represents the amount of time the update algorithm waits before actually skipping this bone's transformation interpolation. In other words, the current level of detail value of each bone will be used as the new time step for updating the that bone's transformation matrix.

Example: Comparing 2 bones:

- Current level of detail of B_1 is 0.048
- Current level of detail of B_2 is 0.032

Notice that at run time, each bone is independent from all other bones, and it will skip its transformation interpolations using its own level of detail value. In the example above, the application is assumed to be running at 60 frames per second, threfore the duration of each frame is 0.016 seconds or 16 milliseconds.

Bone B_1 has a higher priority, and will skip a transformation interpolation each 48 milliseconds, which in this case is each 3 frames, while bone B_2 has a relatively lower level of detail value, and will skip a transformation interpolation each 32 milliseconds, which in this case is each 2 frames.

Note that the level of detail values used in the previous example are very low. The animated model should be really far from the viewer in order to generate level of detail values that low.

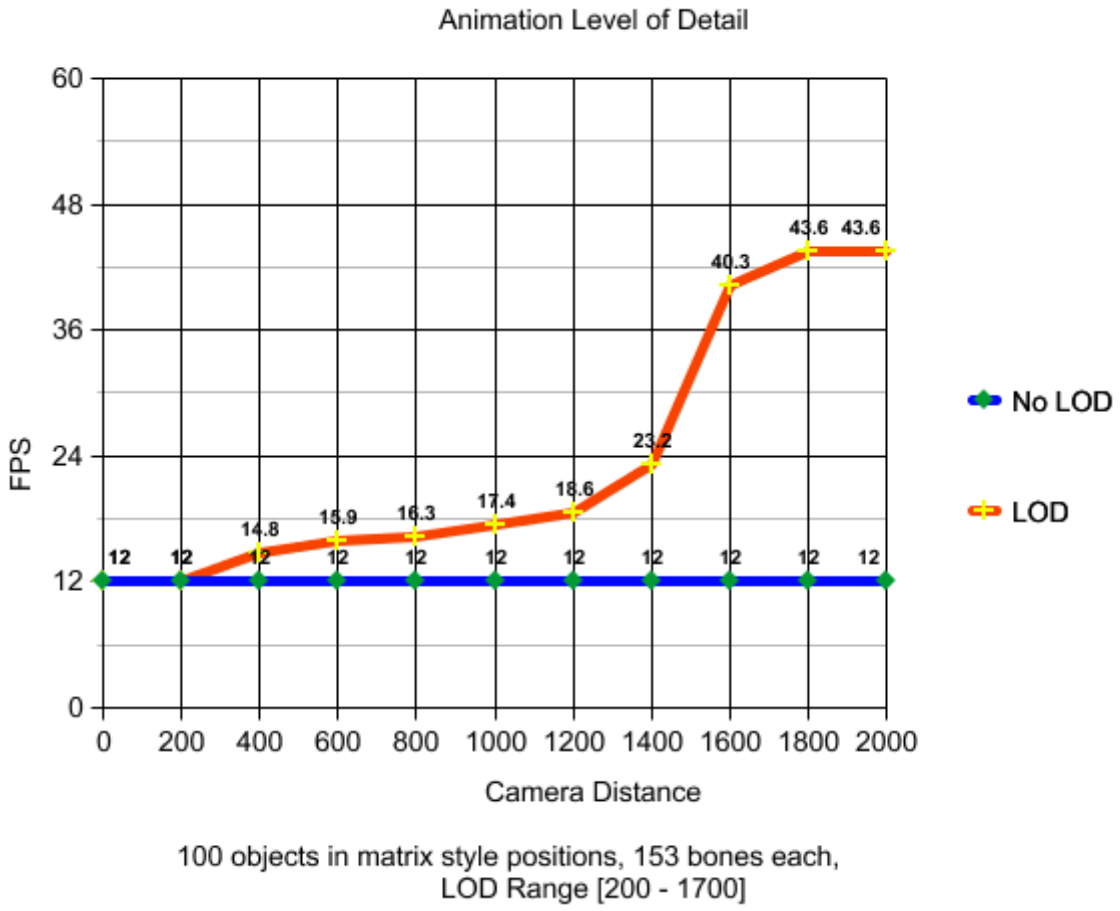
13 Test and benchmarks

The following test compares the frame rate of the application where 100 animated models, each made out of 153 bones, are positioned in a matrix or grid form. The level of detail distance range for this test is [200 – 1700], which means no level of detail will be applied to any model whose distance separating it from the viewer is less than the range's lower bound (200 units). On the other hand, if that distance is greater than the range's upper bound (1700 units), then the model will skip all its transformation interpolations. Finally, if that distance is within the level of detail range, then a dynamic ratio will be computed and multiplied by each bone's current priority in order to generate that bone's current level of detail.

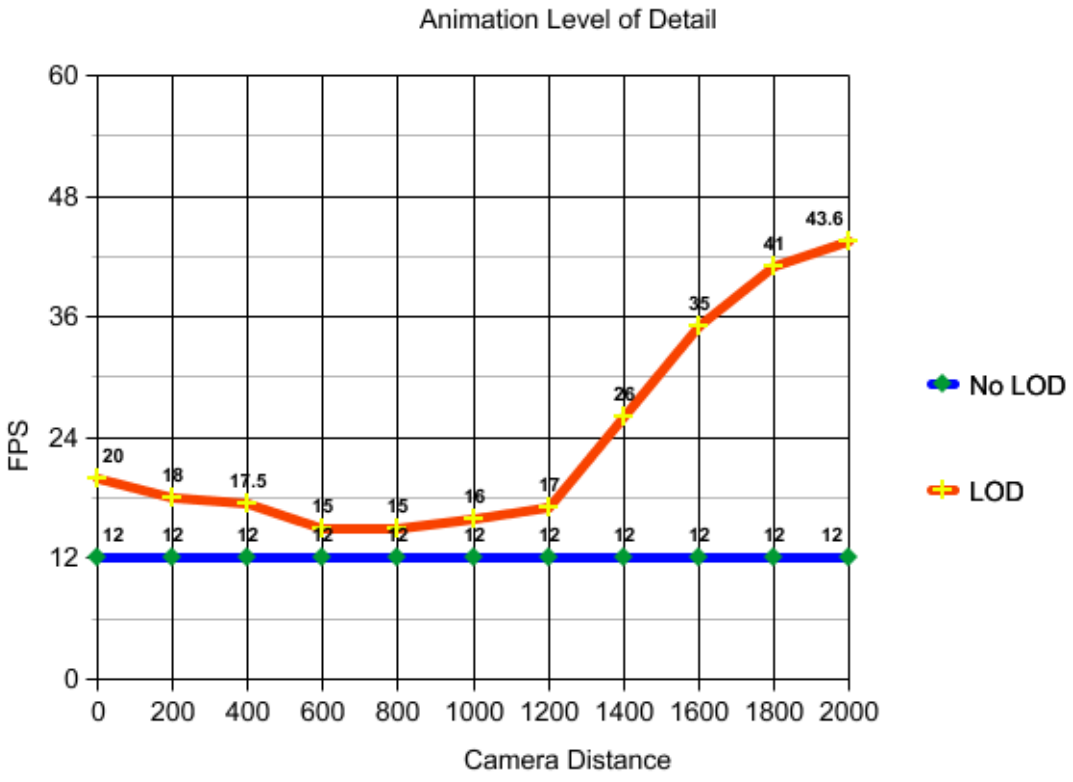
Notice that when no level of detail is applied, the frame rate is always fixed (At 12 frames per second in this test), because each of the 100 animated model is interpolating all its bones' transformations.

But when the application is set to take advantage of the animation level of detail optimization technique, its frame rate starts increasing when the distance separating the camera from the animated models becomes greater than 200, which is logical, since 200 is the lower bound of the level of detail range.

An extreme gain in performance is noticed when the distance separating the camera from the animated objects becomes greater than 1400~1500. The one and only reason behind this fact is that at that distance, some of the objects are beyond the upper bound of the level of detail range, which means all their transformation interpolations are skipped.



The next test uses the same animated model of the first test, which is made out of 153 bones, but the 100 objects are placed in random locations in the scene, which is closer to how they would be in a real game or application. The same LOD range is used, which is [200 – 1700]



100 objects in random positions within a pre-defined cube, 153 bones each, LOD Range [200 - 1700]

We can see that in this formation, the frame rate of the application is always above the normal frame rate where no level of detail is applied. This is due to the fact that some of the animated models are already far from the camera. Notice that the frame rate drops a little bit when the distance separating the camera from the center of the group increases, which is logical since more objects are becoming visible. Animated models skip all their transformation interpolations when they are visible from the view point.

But as soon as that distance increases more and more, we can see the frame rate picking up again, until it reaches its maximum value of 43.6 frames per second, which is identical

to the previous test, because at this distance, all the animated models are skipping all their transformation interpolations

14 Future Work

This level of detail approach for interpolated animations of articulated models presents a technique for reducing the amount of time needed to update animated models with minimal to no visual artifacts.

The decision to skip a bone's transformation interpolation is based on the bone's current priority (since each of the bone's key frames has its own priority depending on the *degree* of visual difference it has compared to the previous key frame). This priority was computed based on 6 different sub priorities which are:

- Translation priority
- Rotation priority
- Scale priority
- Number of children priority
- Depth priority
- User-set priority

These 6 sub priorities are treated equally in the current approach, which means they all contribute to the final priority value using the same weight:

$$w = \frac{1}{6}$$

$$P_{b_i}^{j} = w.P_T + w.P_R + w.P_S + w.P_{Num} + w.P_{Depth} + w.P_{User}$$

This paper also mentions a more adaptive way to compute the final priority value by using a weighted average. In other words, each of the 6 sub priorities will have its own unique weight:

w_T: *Weight of the Translation* sub *priority*
w_R: *Weight of the Rotation* sub *priority*
w_S: *Weight of the Translation* sub *priority*
w_{Num}: *Weight of the number of children* sub *priority*
w_{Depth}: *Weight of the Depth* sub *priority*
w_{User}: *Weight of the User set* sub *priority*

$$P_{b_i}^{j} = w_T . P_T + w_R . P_R + w_S . P_S + w_{Num} . P_{Num} + w_{Depth} . P_{Depth} + w_{User} . P_{User}$$

The 6 different weights could be arithmetically generated by analyzing the animated model. Before computing the priorities of the bones, a new "analysis" pass should be applied to the animated model. This pass should collect information like:

- How much does the bones' translations contribute to the shape of the animated model?
- How much does the bone's rotations contribute to the shape of the animated model?
- How much does the bone's scales contribute to the shape of the animated model?
- How much does the bone's number of children contribute to the shape of the animated model?
- How much does the bone's depth contribute to the shape of the animated model?

Based on this analysis or profiling, the level of detail algorithm should assign a different weight for each of the sub-priorities, each according to how "much" they contribute to the animated model's shape, or in other words, how *important* they are.

This will enhance the adaptivity of this level of detail approach by adding another level of customization for each different animated model.

15 References

- Smooth View-Dependent Level-of-Detail Control and tis Application to Terrain Rendering – Hugues Hoppe – Microsoft Research

- Real-Time, Continuous Level of Detail Rendering of Height Fields – Peter Lindstrom – LarryF.Hodges, David Koller – Nick Faust – William Ribarsky – Gregory A. Turner

- Tunneling for Triangle Strips in Continuous Level-of-Detail Meshes – A. James Stewart

- GLOD: Level of Detail for the Masses – Jonathan Cohen – David Luebke – Nathaniel Duca – Brenden Schibert – Johns Hopkins University – University of Virginia

- Adaptive Real-Time Level-of-Detail-based Rendering for Polygonal Models – Julie C.Xia - Jihad El-Sana – Amitabh Varshney

- Unpopping: Solving the Image-Space Blend Problem for Smooth Discrete LOD Transitions – markus Giegl and Michael Wimmer – Institute of Computer Graphics and Algorithms – Vienna University of Technology

- Automatic Shader Level of Detail – Marc Olano – Bob Kuehne – Maryann Simmons – University of Maryland Baltimore County

- Creation and Control of Teal-time Continuous Level of Detail on Programmable Graphics Hardware – R. Sounthern – J. Gain – University of Cape Town South Africa

- Generating Multiple Level of Detail from Polygonal Geometry Models – G. Schaufler – W. Sturzlinger

- Smooth Levels of Detail – Dieter Schamlstieg – Gernot Schaufler

- LOD Generation with Discrete Curvature Error Metric – S.J. Kim – W.K Jeong – C.H.Kim – Dept. of Computer Science & Engineering – Korea University

- Progressive Buffers View-dependent Geometry and Texture LOD Rendering – Pedro V.Sander – Jason L. Mitchell – Ati Research

- Texturing of Continuous LOD Meshes with the Hierarchical Texture Atlas – Hermann Birkholz – University of Rostock - Germany

- Progressive Meshes – Hugues Hoppe – Microsoft REsearch

- Quality Strips for models with level of detail – Oscar E. Ripolles – Miguel Chover – Fransico Ramos

- Evaluation of Real-Time Continuous Terrain Level of Detail Algorithms

- Real-time Terrain Rendering using Smooth Hardware Optimized Level of Detail

- Continuous Level of detail on Graphics Hardware

- Smooth Geomerty Images - F. Losasso1, H. Hoppe2, S. Schaefer3, and J. Warren3

- Progressive Simplicial Complexes – Jovan Popovic – Hugues Hoppe

- Levels of Detail for Crowds and Groups -C. O'Sullivan†, J. Cassell§, H. Vilhjálmsson§, J. Dingliana.†, S. Dobbyn†, B. McNamee†, C. Peters† and T. Giang†

- Dynamic control of Mesh LODs by Using a Multiresolution Mesh Data Structure – Hongbin Zha – Yoshinobu Makimoto – Tsutomu Hasegawa

- Real-time Mesh Simplification Using the GPU – Christopher DeCoro – Natalya Tatarchuk

- Motion Level-of-Detail A Simplification Method on Crowd Scene – Junghyun Ahn – Kwangyun Wohn

- Large Mesh Simplification using Processing Sequences - M. Isenburg - P. Lindstrom, S.Gumhold - J. Snoeyink

- SUCCESSIVE MAPPINGS: AN APPROACH TO POLYGONAL MESH SIMPLIFICATION WITH GUARANTEED ERROR BOUNDS

- JONATHAN COHEN, DINESH MANOCHA, y AND MARC OLANO z

- SIMULTANEOUS MESH SIMPLIFICATION AND NOISE SMOOTHING OF RANGE IMAGES - D. L. Page, Y. Sun, A. F. Koschan, J. K. Paik, M. A. Abidi

- Mesh Simplication – Akash Kushal

- An Effective Feature-Preserving Mesh Simplification Scheme Based on Face Constriction - Jian-Hua Wua, Shi-Min Hua, Chiew-Lan Taib and Jia-Guang Suna

- An Efficient Mesh Simplification Method with Feature Detection for Unstructured Meshes and Web Graphics - Bing-Yu Chen - Tomoyuki Nishita

- Mesh Simplification using Four-Face Clusters - Luiz Velho

- Mesh Simplification Based on Shading Characteristic - ZHANG Hui – SHU Huazhong - LUO Limin

- Triangle-Mesh Simplification using Error Polyhedra - Mark Eastlick and Steve Maddock

- Mesh Simplification and Adaptive LOD for Finite Element Mesh Generation - Hiroaki Date - Satoshi Kanai - Takeshi Kishinami

- Subdivision Surface Simplification - Won-Ki Jeong - Kolja K¨ahler - Hans-Peter Seidel

- High-Quality Simplification with Generalized Pair Contractions Pavel Borodin - Stefan Gumhold - Michael Guthe - Reinhard Klein

- Simplification of Arbitrary Polyhedral Meshes Shaun D. Ramsey - Martin Bertram - Charles Hansen

- Mesh Reduction with Error Control - Reinhard Klein - Gunther Liebich - W. Straßer

- Surface Simplification Based on a Statistical Approach V. Savchenko - M. Savchenko - O. Egorova - I. Hagiwara

- Level-of-Detail Shaders - Marc Olano - Bob Kuehne