OpenGL:Tutorials:Basic Bones System

From GPWiki

Contents

- 1 What is a bone system
- 2 Why use bones
  - 2.1 Keyframe animation
    - 2.1.1 Advantages
    - 2.1.2 Disadvantages
  - 2.2 Skeletal animation
    - 2.2.1 advantages
    - 2.2.2 Disadvantages
    - 2.2.3 So should I always use bones?
- 3 How does a bone system work?
- 4 The skeletal structure
  - 4.1 Using bones
  - 4.2 Should I use a bone structure or a joint structure?
- 5 Animating?
  - 5.1 Switching between movements
- 6 A simple skeletal system in 2D
  - 6.1 Simple hierarchical structure of bones
  - 6.2 Drawing the skeleton in 2D
  - 6.3 Moving the structure
    - 6.3.1 Moving the puppet around
    - 6.3.2 Selecting a bone
    - 6.3.3 Rotating a bone
    - 6.3.4 Changing length of a bone
    - 6.3.5 Set a degree of freedom for our bones
  - 6.4 Animating
    - 6.4.1 Keyframes
    - 6.4.2 Interpolation
    - 6.4.3 Coding
  - 6.5 Simple polygon skinning
    - 6.5.1 How does a skin work?
      - 6.5.1.1 Skinning with multiple polygons
      - 6.5.1.2 Working on joints
        - 6.5.1.2.1 Parent-side drawing
      - 6.5.1.3 Skinning an entire mesh
- 7 A 3D application using bones
  - 7.1 A check before starting
  - 7.2 Data structures
- 8 Source codes
- 9 External Links

What is a bone system

A bone system - aka skeletal system - is a technique used to create skeletal animations. A skeletal animation consists of a skin mesh and an associated bone structure, so moving a bone will move the associated vertices of the mesh, exactly as happens in reality: we each have a skeletal structure with muscles and skin on it.

Why use bones

Traditional mesh programming uses "3D sprite" animation (or keyframe animation): you have a 3D mesh for each frame: just rendering frame after frame, produces the motion.

Keyframe animation

Advantages

Simple to implement
: given a 3D mesh with framed animation, you just have to load each frame in memory and use a pointer to navigate through the frame

Fast
: once loaded in memory you simply render the mesh of the actual frame: no calculations are needed

Looks good
: Since each model is pre-made, you are sure that the model will look like it did when you created it

Disadvantages

Memory
: Loading a mesh for each frame of the animation uses a lot of memory, especially if there are many different characters in our game, and if there are lots of animations (walking, running, sitting, using, shooting, ...)

No/few interaction
: Interaction with the world is "fake": you must have an animation for each possible interaction and of course you can't apply (or it's complex and slow to implement) physics to your meshes.

Of course if you need an animation for a little bunny in your game - as a non-principal character - which is composed of 3-4 frames, 100 vertices per frame, with one single action (running), using the 3D meshes is a good idea: memory waste isn't high, and since we'll have many bunnies, we prefer to not waste computation time.

But what if you have to write a fighting game (like Soul Calibur, Tekken or Street Fighter)? Yes, the old Street Fighter used 2D sprites to fight, but now, we have a great technique at our disposal: skeletal animation, using bones.

Skeletal animation

advantages

Memory
: This uses very little memory: each character has a single model. Each animation and each action, is a set of moves done by the bones (few vertices) and not by the entire mesh (many vertices).

High interaction
: Once you can master and move each part of the mesh as you like, you can make it interact with the world: kicking a wall, the leg will stop when it touches the wall, since this animation is computed in real-time by the bone system. Of course many other actions are possible: moving the head, dancing,

fighting etc.

Upgradability and reuse
: Imagine you have created a fighting game, each fighter can walk, run and kick. Using skeletons you can export a single mesh for each fighter and a file with movements for each action: walk, run and kick. If you need to add a punch action, you don't have to touch player's meshes: you just have to export another movement file, and all the fighters can use that, without modifying meshes or the game engine.

Disadvantages

Computation time
: Of course, moving a bone and related vertices takes some time that is added to the rendering time. For this reason, bones are a modern technique, not used in the past (in fact I don't think the Street Fighter game used a bone system).

Joints are visible
: A disadvantage in this system is that joints are visible, since the animation is computed in realtime, and of course sometimes we don't have enough processing power to compute complex calculations on the joint.

So should I always use bones?

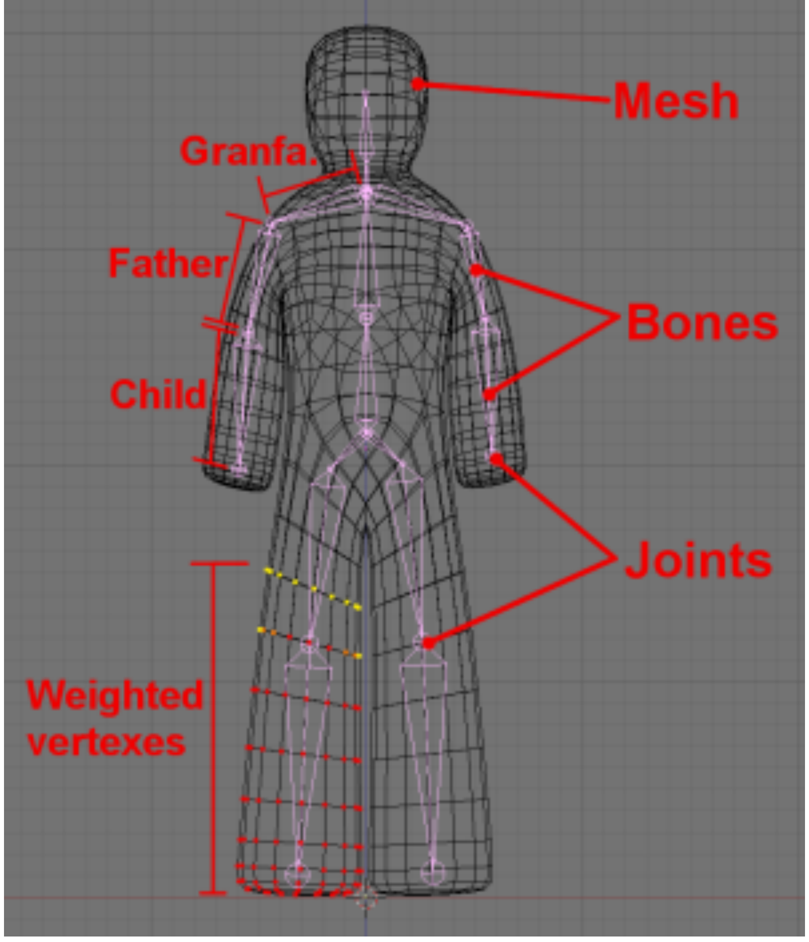
Absolutely not. Sometimes it's far more efficient to use a classic 3D framed model, especially for animations that involve few interactions with the world, or for a great number of mesh instances. Bone systems are great when you need interaction between characters, like FPS or fighting games, or when you want bodies (alive or not) to follow the laws of physics. You can also use this method if you need a lot of control over the animation frames: for example, if you need to implement a "bullet time" effect in your game (like in the movie "The Matrix") you need to slow down your animation, and bones allow you to run your animation smoothly at an arbitrary speed.

How does a bone system work?

This kind of system involves some elements:

1. The object mesh (usually you can use a single mesh, but also more than one)
   - Using weighted vertexes
2. A skeleton for the mesh, composed by a hierarchy of
   - Bones
   - Joints

The mesh is connected with the skeleton. Each vertex of the mesh is associated to a bone (or more than one) and has a weight, which means how much bone movement affects the vertex movement. Each bone in the skeleton is connected to two joints, and each joint is connected to at least one bone. When you have to do an animation, just pre-calculate the key-positions of your skeleton (maybe with a 3D editing program, like Blender or MaYa), then create an algorithm that calculate interpolation between movements. Finally, just apply the interpolation of the vertices basing your calculations on the vertex weights and bone hierarchy.



The act of connecting a mesh to a bone structure is called "skinning". When you apply a skin you should also define vertex weights for each bone. Usually this is done in the 3D modeling software (I love Blender, and it allows you to do this), then you can export the mesh with weighted vertices, bones and textures (if you need to). Sometimes, programs allow you to import a pre-made skeletal structure, then you can build your mesh around it. This is common with highly-customizable games. For example, if you want to build your own character in UT/Quake, just open the default skeletal structure with a modeling program (like MilkShape 3D, which comes with importer/exporter and pre-made skeletal structures), build your model around it, and export the mesh for the game you want to mod. As you can see, it's easy to make new characters with a bone system: just create a mesh and associate the vertices.

The skeletal structure

Mainly the skeletal structure is organized in a hierarchical way: for example femur is the father of tibia and pelvis is father of femur, but basically you can view the skeletal structure as tree.

You can define this tree by using only bones or only joints (maybe also using both, though I see no advantage to doing so)

Using bones

You can define a structure similar to this

```
struct Bone_t
{
   float x1, y1, x2, y2;
   struct Bone_t *father;
};
```

Here a bone is defined like a line, with a father as a pointer of the same type. When a child bone is linked to another, just use the father's (x2, y2) as the starting point of this bone. There are 2 ways to do this

Children coords are offset with respect to the father In this case child.x1 and child.y1 should be both at 0.0. So we can just set an offset like this

```
child.x1 = child.y1 = 0.0;
child.x2 = 10.0;
child.y2 = 15.0;
child.parent = &parent;

/* Now drawing the bones */
child.x1 = child.parent->x2;
child.y1 = child.parent->y2;
child.x2 += child.x1;
child.y2 += child.y2;
draw(parent);
draw(child);
```

Children coords are absolute In this case, child is positioned somewhere on the screen, so a child = {10.0, 11.0, 20.0, 21.0} is a line which goes from (10.0, 11.0) to (20.0, 21.0). Of course this method is Ok if you plan to explode your character in little pieces (so you can easily manage each bone as a different piece), but what if you have to compose bones of a normal skeleton, that is a child bone attached to a parent one? We can make a distinction: bone can be in an absolute position or at a relative position. Just add a flag to our structure that handle this, so

```
struct Bone_t childTemp;

child.x1 = 10.0;
child.y1 = 11.0;
child.x2 = 20.0;
child.y2 = 21.0;
child.parent = &parent;

/* Now drawing the bones */
if (!child.isAbsolutedPositioned) /* This is relative to the parent's position, so translate it */
{
    childTemp.x2 -= child.x1 - child.parent->x2;
    childTemp.y2 -= child.y1 - child.parent->y2;
    childTemp.x1 = child.parent->x2;
    childTemp.y1 = child.parent->y2;
}
else
{
    /* This is already how we want it to be drawn, so do nothing */
}
draw(parent);
draw(childTemp);
```

Of course this calculation must be defined in draw(), but I'm placing it here for clarity.

Anyway, this is only a method to define a bone. Some programmers prefer to use a single coord (x,y) for it's positioning and then an angle and a length relative to the parent - and some use an additional (x2, y2) coord to store the end of the bone, to avoid computing sin() and cos() functions every time a bone is drawn.

Other programmers, don't even use bones. In fact, you can also define joints, and consider a bone as the space between then, but actually without using a bone structure, which is implicit: a bone is between 2 joints, so, defining a bone is like defining 2 joints, a father and a child. The structure for a joint is like this:

```
struct Joint_t
{
    float x, y;
    struct Joint_t *parent;
};
```

Of course, you can implement these structure (both Bone_t and Joint_t) using a pointer to the parent object (as I did) or using a structure for keeping all children, like this:

```
struct Bone_t
{
    /* coords */
    struct Joint_t *children[MAX_CHILDRENS];/* An array to pointers. You can use a null terminating *
                                             /* pointer or just add a count variable */
};
```

And as you might have thought, this way helps a lot for navigating the bone tree (starting from a root to the leaves), depending on your needs, you may also want to define both: pointers to children and a pointer to parent bone/joint.

Should I use a bone structure or a joint structure?

It's your choice, but keep in mind that

- Using bones means you can manage each one as you wish: if you plan to explode your model in little, articulated pieces, it's a good idea to use bones
- Using joints means you have less redundancy of positions: 3 joints are 2 bones, so you have to store only 3 points in memory, since 2 bones have one joint in common. When using a bone structure, you have 4 points in memory for 2 bones. Hence using joints saves memory, but it's ok only for connected bones.
- If you want to create a bone structure for a sea star (or a structure where there are bones of the same level), it's easy to do with only joints: one father joint is the center of the star. If you are using a bone structure you should create a "null bone", of length and angle 0, which works as a joint.

Animating?

At this point now you should have a base about what a bone system is and what are it's capabilities. Before proceeding in creating a working system, there is another point to discuss: animation, of course.

Animating this kind of system can be done with 2 different techniques, but we'll talk about them later. Now it's important to understand the concept in common in both techniques: an animation is made of **keyframes** and **interpolation**. As we said before using bones helps us to have control over each movement: since moving a bone corresponds to moving an entire set of weighted vertexes, and since we can handle each single bone, it's logical that we can produce frames with an arbitrary sample (that is: calculating 100 frames at 100fps looks like calculating 50 frames at 50fps, but with doubled sampling rate) - this is not possible (or it's difficult) with 3D sprites.

Usually you'll want to use a precalculated animation: you have a file containing a finite set of frames representing the - for example - walking cycle. Each of these frames is called a *keyframe* and represents a key position for each bone at a specific time. Since we have keyframes, and we need a virtually infinite set of frames, we need to use interpolation. Interpolation is (from wikipedia) "a method of constructing new data points from a discrete set of known data points". Our set is the keyframes taken from the file. Actually this means that if you have a point at location (x1, y1) at time 0, and it is at (x2, y2) at time 10, interpolation can calculate (with a good approximation) all the positions of the points when time is between 0 and 10.

In our case we'll use the linear interpolation for calculating the position of the bones in frames that aren't keyframes.

Now that you know basically how an animation is done, let me explain the two techniques used in creating an animation.

1. Forward kinematic
2. Inverse kinematic

Imagine you have an articulated puppet just like a drawing puppet or a GI Joe. Now let's try make to it walk (actually we don't want to move it in realtime, just calculate each frame of the animation). If you use a forward kinematic, starting from a stand position, you have to rotate it's femur and it's tibia a little, then in the next frame you rotate them more, then it must extend it's leg to make a step, then it falls on a foot and the step is done. Doing this for both legs creates the walking cycle

Then you can make it walk using the inverse kinematic. Just take its foot in your hands, and move the foot just like they would move in a walking cycle. The legs will follow the feet, and the animation cycle is done.

(an image which explains the two concepts)

Actually, FK is the one applied in a real body: each muscle rotates a bone of some degree respecting the parent bone - which muscles are fixed to. But as you can see, IK is much more easy to use, since you just have to say: "I want this foot here", and the leg will move properly.

As you can imagine, IK needs something to work as it should: a degree of movement and min/max degrees for each articulation. For example, a knee can't be opened more than 180°, and can't be closed less then 20° or 30° (these are max/min degrees for the knee), and you also know that a knee has only one degree of movement: you can only open and close it. Some other articulations, have more degrees of movement, for example a shoulder has 3 degrees of movement, since you can rotate an arm around the X Y or Z axes (you can throw your arms up or down, you can open or close your arms as when you embrace something, and you can rotate an arm on it's axis).

FK doesn't need to know the degrees of movement for each articulation, since you move each bone as you wish.

In both cases, we have to store some positions in each frame, and interpolate them to create an animation.

Switching between movements

As I said bones have several advantages over static meshes. One of these advantages is not really particular, but it's - graphically speaking - important. For example: your player is running, then, in the middle of the running cycle, you stop pressing the "run" button, and your player starts to walk, immediately. In reality, this never happens: when you pass from a running state to a walking state, you have to slow down to decrease your speed, and then you start to walk, naturally interpolating your actions. This seems normal to us, but it's actually hard to do in video games. In 2D and 3D games, when you use sprites instead of a bone system, this interpolation is quite difficult to realize, and usually this involves a lot of work from the graphic designers, since they have to draw all the interpolating movement between actions, and this takes lots of time and memory. If instead of sprites you use a bone system, this issue is far more simple to solve, without needing lots of work with graphics. Since you can handle each bone and interpolate the bone positions between key frames, why not interpolate between different actions, instead of just different frames? Infact, this is the solution to our problem: just interpolate different animations. In this way you can easily switch between running and walking, without producing "time-zero" movements.

Of course if you want to get better results, defining an additional animation cycle can help the movement: if you want to jump while running, the jumping animation while running is quite different than a jump while walking, and it's different than a jump when you aren't moving.

A very good example of this "action interpolation" can be seen in the newest fighting games, like Soul Calibur 2, Tekken or Dead Or Alive 3. Another very good example that comes in my mind - is in the Metroid Prime series, from Nintendo: Samus can morph herself in a sphere, and you can do this while running or walking, and IMHO the animation interpolation is done perfectly. Also soccer games like PES or FIFA must have good interpolation between actions. For a comparison, just look at these examples and then take a look at a fight in an old game like Street Fighter 2, where the "trick" to avoid a "time-zero" movement, was to execute actions sequentially, to avoid the necessity of interpolating animations (so for example, sequences like "run-jump-run" becomes like "run-stop-jump forward-stop-run").

A simple skeletal system in 2D

Now you should have a good background about skeletal animation. Now let's design a basic work, in two dimensions (for simplicity), to understand how bones work and to try our hand at this kind of programming.

The system that follows is simple, actually used to learn how to program and manage bones. We'll follow a time line, starting from a simple example to go toward something "more difficult"

1. Simple hierarchical structure of bones
   - Here we define the structures, how the tree is composed, we define the functions and the structure to manage our skeleton
2. Drawing the skeleton in 2D
   - Then we define functions to draw the bones on the screen, using OpenGL in ortho view.
3. Moving the structure
   - Next we give users control over the bones: select a bone and move it
4. Animating
   - Then we start an animation: given an animation cycle, we play it.
5. Simple polygon skinning
   - When we have a simple bone system, we want to put some skin on it. We start by attaching some separate polygons.
6. Skinning a single mesh
   - Finally we attach one single mesh on the bones, and we can apply to them any single mesh.

Simple hierarchical structure of bones

First of all let's create the Bone structure (yes, I prefer to use Bones instead of Joints, only because it sounds easy :D). And in this example, I prefer to use a bone calculated by angle and length, instead of 2 points in the space. This structure must contain:

- Coordinates of the starting point
- An angle
- A length
- Flags to determinate if the bone is in absolute position or not
- Hierarchy data (children and parent)
- A name, just for the sake of the example (of course you don't need this in a game)

So, let's define the structure:

```c
/* C code, made for tabs of 8 spaces
 * uint8_t is defined in the standard C header stdint.h
 */

/* Define numbers and flags */
#define MAX_CHCOUNT                     8       /* Max children count */
#define BONE_ABSOLUTE_ANGLE             0x01    /* Bone angle is absolute or relative to parent */
#define BONE_ABSOLUTE_POSITION          0x02    /* Bone position is absolute in the world or relative
#define BONE_ABSOLUTE                   (BONE_ABSOLUTE_ANGLE | BONE_ABSOLUTE_POSITION)

typedef struct _Bone
{
        char name[20];                          /* Just for the sake of the example */
        float x,                                /* Starting point x */
              y,                                /* Starting point y */
              a,                                /* Angle, in radians */
              l;                                /* Length of the bone */

        uint8_t flags;                          /* Bone flags, 8 bits should be sufficient for now */
        uint8_t childCount;                     /* Number of children */

        struct _Bone *child[MAX_CHCOUNT],       /* Pointers to children */
                     *parent;                   /* Parent bone */
} Bone;
```

After this of course we need to define some functions to handle bones. Actually I want to define these functions

boneAddChild
: Takes a bone and adds a child. If bone doesn't exists, this should create it (without a child)

boneDumpTree
: Takes a bone and print on stdout the hierarchy

boneLoadStructure
: Takes a file path and loads the bone tree from it

boneFreeTree
: Takes a bone and clears the memory

```c
/* Create a bone and return it's address */
Bone *boneAddChild(Bone *root, float x, float y, float a, float l, Uint8 flags, char *name)
{
        Bone *t;
        int i;

        if (!root) /* If there is no root, create one */
        {
                if (!(root = (Bone *)malloc(sizeof(Bone))))
                        return NULL;
                root->parent = NULL;
        }
        else if (root->childCount < MAX_CHCOUNT) /* If there is space for another child */
        {
                /* Allocate the child */
                if (!(t = (Bone *)malloc(sizeof(Bone))))
                        return NULL; /* Error! */

                t->parent = root; /* Set it's parent */
                root->child[root->childCount++] = t; /* Increment the childCounter and set the pointer
                root = t; /* Change the root */
        }
        else /* Can't add a child */
                return NULL;

        /* Set data */
        root->x = x;
        root->y = y;
        root->a = a;
        root->l = l;
        root->flags = flags;
        root->childCount = 0;

        if (name)
                strcpy(root->name, name);
```

```
	else
		strcpy(root->name, "Bone");

	for (i = 0; i < MAX_CHCOUNT; i++)
		root->child[i] = NULL;

	return root;
}

/* Free the bones */
Bone *boneFreeTree(Bone *root)
{
	int i;

	if (!root)
		return;

	/* Recursively call this function to free subtrees */
	for (i = 0; i < root->childCount; i++)
		boneFreeTree(root->child[i]);

	free(root);

	return NULL;
}

/* Dump on stdout the bone structure. Root of the tree should have level 1 */
void boneDumpTree(Bone *root, Uint8 level)
{
	int i;

	if (!root)
		return;

	for (i = 0; i < level; i++)
		printf("#"); /* We print # to signal the level of this bone. */

	printf(" %4.4f %4.4f %4.4f %4.4f %d %d %s\n", root->x, root->y,
	       root->a, root->l, root->childCount, root->flags, root->name);

	/* Recursively call this on my children */
	for (i = 0; i < root->childCount; i++)
		boneDumpTree(root->child[i], level + 1);
}
```

Now that we have these 3 simple functions to handle bones, let's try them! I create this main function

```
int main(int argc, char **argv)
{
	Bone *root,
	     *tmp,
	     *tmp2;

	int i;

	/* Create a root bone
	 * this is a "null" bone which represent a single point, which is the center of the structure.
	 * Do you remember the sea star example above?
	 */
	if (!(root = boneAddChild(NULL, 100, 100, 0, 0, 0, "NullBone")))
	{
		fprintf(stderr, "Error! Can't create a root!\n");
		exit(EXIT_FAILURE);
	}

	/* Creating a bone which has (x,y) == (0,0) and BONE_ABSOLUTE_POSITION NOT set
	 *  causes this bone to start where its parent ends.
	 * If ABSOLUTE_POSITION is off, x and y work as offsets with respect to the parents end positi
	 * If it's on, then (x,y) will be placed at an absolute position on the screen.
	 */
	boneAddChild(root, 100, 100, M_PI_2, 10, BONE_ABSOLUTE, "Head");
	tmp = boneAddChild(root, 0, 0, -M_PI_2, 30, 0, "Back");
	tmp2 = boneAddChild(tmp, 0, 0, -M_PI_4, 30, 0, "LLeg");
	boneAddChild(tmp2, 0, 0, 0, 30, 0, "LLeg2");
	tmp2 =  boneAddChild(tmp, 0, 0, -2 * M_PI_4, 30, 0, "RLeg");
	boneAddChild(tmp2, 0, 0, 0, 30, 0, "RLeg2");
	tmp = boneAddChild(root, 0, 0, 0, 20, 0, "LArm");
	boneAddChild(tmp, 0, 0, 0, 20, 0, "LArm2");
```

```
        tmp = boneAddChild(root, 0, 0, M_PI, 20, 0, "RArm");
        boneAddChild(tmp, 0, 0, M_PI, 20, 0, "RArm2");

        boneDumpTree(root, 0);

        root = boneFreeTree(root);

        return EXIT_SUCCESS;
}
```

Then, to facilitate the creating of bone structures, I created (and it took about 3 hours! [Difficult]) this function which loads a skeletal structure from a file. The file is in the same format outputted by the dump function above, which is:

1. Level made of #s, starting from 1 which is the root
2. (x, y) coords, angle and length
3. Flags
4. Name

Here some examples of these files

A star

```
#    0    0      0     0    5    3   Root
##   0    0 0.000   100    0    0   One
##   0    0 1.256   100    0    0   Two
##   0    0 2.512   100    0    0   Three
##   0    0 3.768   100    0    0   Four
##   0    0 5.024   100    0    0   Five
```

A snake?

```
# 0.0000 0.0000 1.0000 50.0000 1 3 Root
## 0.0000 0.0000 1.0000 50.0000 1 0 One
### 0.0000 0.0000 1.0000 50.0000 1 0 Two
#### 0.0000 0.0000 1.0000 50.0000 1 0 Three
##### 0.0000 0.0000 1.0000 50.0000 1 0 Four
###### 0.0000 0.0000 1.0000 50.0000 0 0 Five
```

A sort of human

```
# 0.0000 0.0000 0.0000 0.0000 4 0 Root
## 0.0000 0.0000 1.5708 30.0000 0 0 Head
## 0.0000 0.0000 -1.5708 50.0000 2 0 Back
### 0.0000 0.0000 -0.7854 50.0000 1 0 LLeg
#### 0.0000 0.0000 0.7854 50.0000 0 0 LLeg2
### 0.0000 0.0000 0.7854 50.0000 1 0 RLeg
#### 0.0000 0.0000 -0.7854 50.0000 0 0 RLeg2
## 0.0000 0.0000 -0.1000 40.0000 1 0 LArm
### 0.0000 0.0000 0.1000 40.0000 0 0 LArm2
## 0.0000 0.0000 3.2416 40.0000 1 0 RArm
### 0.0000 0.0000 -0.1000 40.0000 0 0 RArm2
```

This is the function to read these files

```
Bone *boneLoadStructure(char *path)
{
        Bone *root,             /* The root of the tree to load */
             *temp;             /* A temporary root */

        FILE *file;             /* File to load */
        float x,                /* Bone data */
              y,
              angle,
              length;

        int depth,              /* Depth retrieved from file */
            actualLevel,        /* Actual depth level */
            flags;              /* Bone flags */

        char name[20],          /* Buffers for strings */
             depthStr[20],
             buffer[512];

        if (!(file = fopen(path, "r")))
        {
                fprintf(stderr, "Can't open file %s for reading\n", path);
                return NULL;
        }

        root = NULL;
        temp = NULL;
        actualLevel = 0;

        while (!feof(file))
        {
                /* Read a row from the file (I hope that 512 characters are sufficient for a row) */
                fgets(buffer, 512, file);

                /* Get the info about this bone*/
                sscanf(buffer, "%s %f %f %f %f %d %s\n", depthStr, &x, &y, &angle, &length, &flags, na

                /* Avoid empty strings, but this is ineffective for invalid strings */
                if (strlen(buffer) < 3)
                        continue;

                /* Calculate the depth */
                depth = strlen(depthStr) - 1;
                if (depth < 0 || depth > MAX_CHCOUNT)
                {
                        fclose(file);
                        fprintf(stderr, "Wrong bone depth (%s)\n", depthStr);
                        return NULL;
                }

                /* If actual level is too high, go down */
                for (; actualLevel > depth; actualLevel--)
                        temp = temp->parent;

                /* If no root is defined, make one at level 0 */
                if (!root && !depth)
                {
                        root = boneAddChild(NULL, x, y, angle, length, flags, name);
                        temp = root;
                }
                else
                        temp = boneAddChild(temp, x, y, angle, length, flags, name);

                /* Since the boneAddChild returns child's address, we go up a level in the hierarchy *
                actualLevel++;
        }

        fclose(file);

        return root;
}
```

Drawing the skeleton in 2D

Now we should have working code which can create bones and a tree of bones, load skeletons from text files and free memory. I think it's a waste of time to do more: since this work is done for video games, let's draw the skeleton on the screen! For this I'm using SDL and OpenGL: simple, fast and cross-platform. Before starting the code, let's see how we can draw this structure:

- Since each bone can be absolute or relative, we need to manage matrices to save/restore the status before a drawing
    - If a bone is relative to the parent, we have just to glTranslate it to draw the point and glRotate the current matrix.
    - If a bone is absolute, we need to glPushMatrix, and when we need to draw another absolute bone, we need to glPopMatrix.

As you can see, using angles and lengths is useful when drawing with OpenGL.

Let's create the function boneDraw

```
/* TODO: Actually this doesn't handle absolute bones */
void boneDraw(Bone *root)
{
	int i;

	glPushMatrix();

	/* Draw this bone
	 * 1. Translate to coords
	 * 2. Rotate the matrix
	 * 3. Draw the line
	 * 4. Reach the end position (translate again)
	 */
	glTranslatef(root->x, root->y, 0.0);
	glRotatef(RAD2DEG(root->a), 0.0, 0.0, 1.0);

	glBegin(GL_LINES);

	glColor3f(1.0, 0.0, 0.0);
	glVertex2f(0, 0);
	glColor3f(0.0, 1.0, 0.0);
	glVertex2f(root->l, 0);

	glEnd();

	/* Translate to reach the new starting position */
	glTranslatef(root->l, 0.0, 0.0);

	/* Call function on my children */
	for (i = 0; i < root->childCount; i++)
		boneDraw(root->child[i]);

	glPopMatrix();
}
```

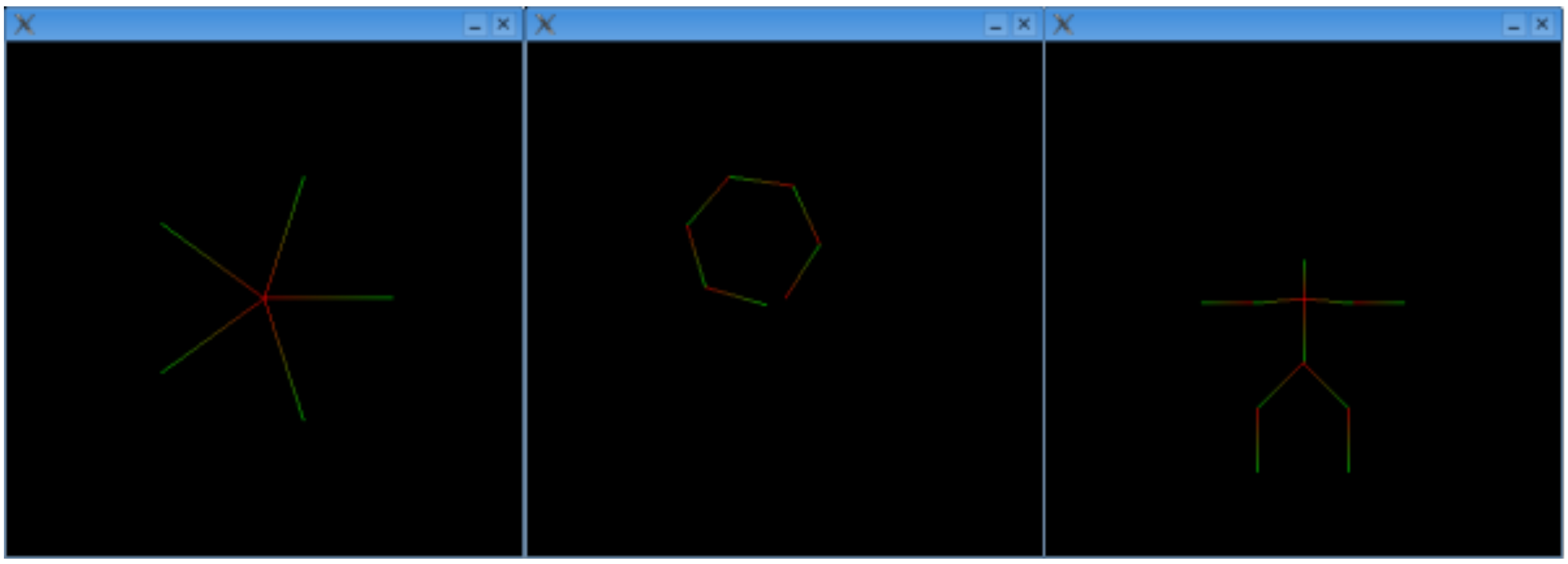
And now let's create a new main to

1. Start SDL and OpenGL
2. Load a skeleton file
3. Draw it on the screen

```
int main(int argc, char **argv)
{
	SDL_Event sdlEv;
	Uint32 sdlVideoFlags = SDL_OPENGL;
	Uint8 quit;

	/* We need one parameter: the structure file */
	if (argc < 2)
	{
		fprintf(stderr, "This program require a filename as parameter\n");
		return EXIT_FAILURE;
	}

	/* Initialize */
	if (SDL_Init(SDL_INIT_VIDEO) < 0)
	{
		fprintf(stderr, "SDL_Init: %d\n", SDL_GetError());
		exit(EXIT_FAILURE);
```

```
        }
        atexit(SDL_Quit);

        /* Start graphic system with OGL */
        SDL_GL_SetAttribute(SDL_GL_RED_SIZE, 5);
        SDL_GL_SetAttribute(SDL_GL_GREEN_SIZE, 5);
        SDL_GL_SetAttribute(SDL_GL_BLUE_SIZE, 5);
        SDL_GL_SetAttribute(SDL_GL_DEPTH_SIZE, 5);
        SDL_GL_SetAttribute(SDL_GL_DOUBLEBUFFER, 1);

        if (!SDL_SetVideoMode(400, 400, 0, sdlVideoFlags))
        {
                fprintf(stderr, "SDL_SetVideoMode: %s\n", SDL_GetError());
                exit(EXIT_FAILURE);
        }

        glShadeModel(GL_SMOOTH);
        glViewport(0, 0, 400, 400);

        glMatrixMode(GL_PROJECTION);
        glOrtho(-200, 200, -200, 200, -1, 1);
        glMatrixMode(GL_MODELVIEW);

        /* Application Initialization */
        Bone *root, *p;
        int i;
        root = boneLoadStructure(argv[1]);

        /* Main loop */
        quit = 0;
        while (!quit)
        {
                while (SDL_PollEvent(&sdlEv))
                        switch (sdlEv.type)
                        {
                                case SDL_QUIT:
                                        quit = 1;
                                        break;

                                default:
                                        break;
                        }

                glClear(GL_COLOR_BUFFER_BIT);
                glLoadIdentity();

                boneDraw(root);

                SDL_GL_SwapBuffers();
        }

        return EXIT_SUCCESS;
}
```

Actually this code, applied to the three skeletons posted above (star, snake? and the humanoid), produces this result:



Moving the structure

Once we have a puppet on the screen, it can be useful. If you are a programmer that can't wait to know everything before starting to code, maybe you already used it to create an animation (and it's possible, using frames, even without interpolation).

Yes, you can do many things with just a bone, but if you have a puppet, you can do more knowing how to move it. And this is exactly what we'll do in the next step.

Actually moving is simple (both in 2D and 3D), thanks to the hierarchy. I start by defining what we want to do with our puppet

- Move the puppet around
- Select a bone
- Rotate a bone
- Change the length of a bone
- Set a degree of freedom for our bones

These are all easy things to do, so let's get started!

Moving the puppet around

Moving is the easiest step (and of course important since usually you want to move objects and characters in your game ;): we have a hierarchal structure, so moving the root bone (which is absolute) will move each child which is relatively positioned. Since root is usually the topmost bone in the hierarchy, we don't have to locate it, just use the root pointer of our code.

I want to click on the window with the left button of the mouse to move it in that position. To do this the code must handle the mouse event, then we can use the root pointer to access the root bone and change the puppet position. Here's the code, which is short and goes in the SDL event handling loop:

```
switch (sdlEv.type)
{
        ...

        case SDL_MOUSEBUTTONDOWN:
                if (sdlEv.button.button == SDL_BUTTON_LEFT)
                {
                        /* We have to translate the click since the
                         *  (0, 0) point is in the middle of the screen
                         * and we have to flip Y coords because in SDL
                         *  it grows inversely to the OpenGL
                         */
                        root->x = (float)sdlEv.button.x - 200.0;
                        root->y = 200.0 - (float)sdlEv.button.y;
                }
                break;

        ...
}
```

Finish :D It's simple, right? And it works (at least it works here :D). You can already animate your puppet using frames, now that you know how to move it in the space.

Selecting a bone

Before rotating a bone, you need to select it. The way it's selected isn't important itself, but there are some things you must know

- Each bone must be uniquely identified in the hierarchy

Selecting is an operation on the tree, so there are many ways to find your bone. Actually I use a very simple recursive function (which isn't optimized for speed. If you are interested in high-speed you must know very well the tree data structure, and this is not the topic), which returns a pointer to the bone. In this tutorial - for now - we use a unique name to identify a bone, this is for clarity when you dump the bones, but actually in your game you may prefer to use a numeric ID for each bone - which is faster and require less memory.

We define a selecting function which, given the bone name, returns a pointer to that bone. The code is simple as you can see

```
Bone *boneFindByName(Bone *root, char *name)
{
	int i;
	Bone *p;

	/* No bone */
	if (!root)
		return NULL;

	/* Check this name */
	if (!strcmp(root->name, name))
		return root;

	for (i = 0; i < root->childCount; i++)
	{
		/* Search recursively */
		p = boneFindByName(root->child[i], name);

		/* Found a bone in this subtree! */
		if (p)
			return p;
	}

	/* No such bone */
	return NULL;
}
```

But to be honest - since using names is a little complex compared to using numbers - we also need something to help us list each bone, so that we can always know the previous and the next bone in the tree. I created this function, which fills a NULL terminated array of strings.

```
#define MAX_BONECOUNT			20

void boneListNames(Bone *root, char names[MAX_BONECOUNT][20])
{
	int i,
		present;

	if (!root)
		return;

	/* Check if this name is already in the list */
	present = 0;
	for (i = 0; (i < MAX_BONECOUNT) && (names[i][0] != '\0'); i++)
		if (!strcmp(names[i], root->name))
		{
			present = 1;
			break;
		}

	/* If itsn't present and if there is space in list */
	if (!present && (i < MAX_BONECOUNT))
	{
		strcpy(names[i], root->name);
```

```
                if (i + 1 < MAX_BONECOUNT)
                        names[i + 1][0] = '\0';
        }

        /* Now fill the list with subtree's names */
        for (i = 0; i < root->childCount; i++)
                boneListNames(root->child[i], names);
}
```

And then I created and filled that array in main

```
char names[MAX_BONECOUNT][20];
names[0][0] = '\0';

boneListNames(root, names);

for (i = 0; (i < MAX_BONECOUNT) && (names[i][0] != '\0'); i++)
        printf("Bone name: %s\n", names[i]);
```

Now that we can actually find a bone by name and we know all the bones, just handle with SDL two keys that allow us to loop through the bones. I'll use N(ext) and P(rev) keys of the keyboard to be vim editor friendly. If you want, you can also use the mouse wheel.

```
        /* As global */
        char *currentName = NULL;

        /* In main */
        int nameIndex = 0;
        /* Calling boneListNames */
        currentName = names[nameIndex];

        /* In SDL event handling loop */
        case SDL_KEYDOWN:
                switch (sdlEv.key.keysym.sym)
                {
                        case SDLK_n:
                                if ((nameIndex < MAX_BONECOUNT) && (names[nameIndex][0] != 0))
                                        nameIndex++;
                                else
                                        nameIndex = 0;
                                break;

                        case SDLK_p:
                                if (nameIndex > 0)
                                        nameIndex--;
                                break;

                        default:
                                break;
                }
                currentName = names[nameIndex];
                break;
```

And of course, it would be nice to add some color to our drawing, so that when a bone is selected it is drawn with different colors, so using the currentName value, we modify the boneDraw function

```
...
glBegin(GL_LINES);

if (!strcmp(root->name, currentName))
        glColor3f(0.0, 0.0, 1.0);
else
        glColor3f(1.0, 0.0, 0.0);

glVertex2f(0, 0);

if (!strcmp(root->name, currentName))
        glColor3f(1.0, 1.0, 0.0);
else
        glColor3f(0.0, 1.0, 0.0);
```

```
glVertex2f(root->l, 0);

glEnd();
...
```

Now everything should be ok. As you see, this is a long process, but not to complex. Of course it's better to use numbers as bone's ID, so that you don't need to keep track of the current bone name and you don't need a function to list all the names - just get the min and max ID in the tree.

Rotating a bone

Well, this is the hot point: rotating a bone is the most common action when handling a puppet, so this is import, but it's very simple. Since we have the name of the currently selected bone, we just have to use the boneFindByName function to retrieve a pointer to that bone, then just modify it's angle value.

This is done again in the event loop, with very few lines of code. To change the angle i use the left and right buttons of the keyboard.

```
case SDLK_LEFT:
        p = boneFindByName(root, currentName);
        if (p)
                p->a += 0.1;
        break;

case SDLK_RIGHT:
        p = boneFindByName(root, currentName);
        if (p)
                p->a -= 0.1;
        break;
```

Changing length of a bone

Like above, this is a simple action: bind two keys for this function (I use up and down), then get the current bone and change the bone's length. The code is simple, as always ;)

```
case SDLK_UP:
        p = boneFindByName(root, currentName);
        if (p)
                p->l += 0.1;
        break;

case SDLK_DOWN:
        p = boneFindByName(root, currentName);
        if (p)
                p->l -= 0.1;
        break;
```

Set a degree of freedom for our bones

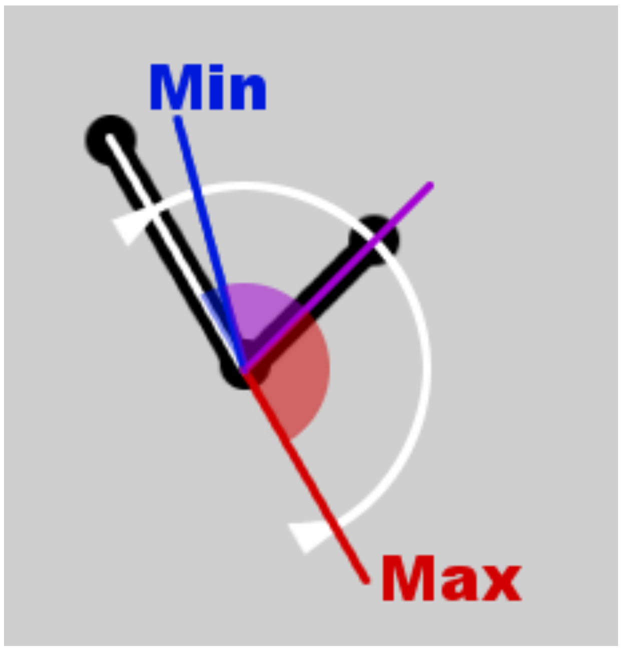
Now, let's add something to our bones: a degree of freedom. This is not really useful if the animation is done with Forward Kinematic, since you have to move each bone - which you can do with the command we just put in the game. If you are using inverse kinematic, or if the user has control of the bones, you may want to add some control to the bone rotations. So, we just have to specify a limit for the angle. We just have to put:

```
float minAngle, maxAngle;
```

in the bone structure, and then check their values before changing the bone angles.

This is really simple, you could probably do it by yourself or just look at the tutorial sources (TODO when they're ready ;)

Below you can see an image showing a limited movement of a 2D bone



Animating

Animating means "giving life". So, let's give life to our skeletal structure :) As I said before, we need two basic concepts:

- Keyframes
- Interpolation

Keyframes

A keyframe literally means a frame which contains a key. Well, suppose you have a time line for your animation: a keyframe represents a special key where the animation changes in a particular way. For example, you want to animate the weaving action, with your hand open that is moved from left to right. The first keyframe (let's say the frame number 0) is your hand on the left which starts to move toward the right. The second keyframe (frame number 5) is your hand on the right, and the last keyframe (frame number 10) is again your hand on the left. As you can see, these frames are "key" since at these points the movement changes radically: between frames 0 and 1, the body status changes from stopped to in movement position. Between frames n and n+1, with 1 <= n < 4 the animation don't change so much, because if you take frame 1 and frame 2, the hand is a little moved to the right. Same with frames 3, 4 and 4, 5. Then, with frames 5 and 6, there is another change in the animation, since your hand stops to move toward right and starts to move to left. And finally at frame 10, the animation changes again because your hand stops.

So, a keyframe is a particular status at time T in your animation, which means a particular change in the animation course.

In our code, keyframes can be implemented using a Keyframe structure, which keeps track of bone status at a specific time. An animation will be just an array of keyframes:

```
typedef struct
{
        uint32_t time;
        float angle[MAX_BONECOUNT],
                                  length[MAX_BONECOUNT];
  } Keyframe;
```

Where

- time is an hi-resolution (millisec) counter that tell us at which time the keyframe occur.
- angle contains the actual angle for each bone in out skeleton
- length contains the length for each bone in our structure

Of course, if your animation don't allow stretching of bones, you can omit the length array to avoid waste of memory. Also, you may prefer to use lists or other forms of data storage instead of arrays, if your game allows high number of bones but lots of models don't reach that number - for instance: if you have 1 human in your game which have 100 bones, and you have 200 enemies which uses 10 bones each, using a float[100] is a great waste of memory since 200/201 characters uses only 10/100 fields of the array. In this case you can use a linked list like this:

```
  typedef struct _KeyData
  {
         float angle, length;
         struct _KeyData *next;
  } KeyData;

  typedef struct
  {
         uint32_t time;
         KeyData *first;
  } Keyframe;
```

Which is a good solution. Of course in this tutorial I won't use this to keep code simple, also because I intend to use only a skeleton at time in the application, and since - also having some skeletons at the same time - the memory waste is acceptable.

Some programmers also like to link each keyframe to a bone, like this:

```
typedef struct
{
        uint32_t time;
        float angle, length;
} Keyframe;

typedef struct
{
        uint32_t id;
        float x, y, a, l;

        Keyframe *animation;
} Bone;
```

This is a good technique, since it allow to:

Use bone-related keyframes, and not skeletal-related
: I take as example the waving hand: you have - just to say - 6 bones: 1 arm bone and 5 bones for fingers. If your animation don't move fingers, using this technique avoid you to save, at each keyframe, status for each bone, instead of the entire skeleton. So, in our case, each finger will be a fixed angle value and don't have related Keyframes, while the arm bone will have 3 keyframes.

Speed up the calculations
: When a keyframe happen, you don't have to refresh the whole tree (in our example you have only to refresh the arm bone), and when you have to refresh each bone, this allow you to doing this in the drawing routine, without involving an additional find-and-refresh function, that could make the code

run slower.

So, it's a good idea to use keyframes like this, since the previous model is good only for explanations :D But if you find good reasons to use the skeletal-related model, please tell us!

Also remember that you may want to change vertexes positions during the animation, so in your Keyframe structure, in addition to angle and length fields, you may want to modify the coords values. The procedure is always the same, I'm just showing it for angles and lengths, it's up to you to understand what fields you need in your application.

Interpolation

Before I said that with boned animation, you don't have to store all the frames of your animation, but just the keyframes. Alright, now we know what a keyframe is, but the question is: if I've only keyframes, how can I animate the skeleton? The answer is: calculating the other frames using interpolation (or linear interpolation, to be more precise).

Keyframes represent a big change in our animation, but other frames don't! This mean that we can guess - knowing some starting conditions - frames between two keyframes. Our guess is done by interpolating the keyframes.

Different interpolation algorithms exist, and in some cases you may prefer to use one instead of another. For example, if you know that your animation starts slow to get faster, you may prefer to use a exponential interpolation, or if you know that your animation starts slow, get velocity and then return to get slower, you may want to use a sine interpolation. But in the common case, we prefer to use linear interpolation, that means that our animation don't change speed: waving an hand with linear interpolation produces that at every frame our hand will rotate for X degrees, and between frames X don't change. To be more precise, a *real* waving action will be more like a sine interpolation, since our hand tend to accelerate and decelerate when we change direction - since our hand have some mass that we have to contrast with our muscles.

In this tutorial - and most of games - linear interpolation is good: it provides a good approximation and a good executing speed.

Linear interpolation is quite simple to realize, here a very simple example:

- Imagine you have a bone B, starting with angle a=0° at time t=0.
- The animation of B ends on time t=10 and angle a=180°.
- The animation runs at 12fpt (frame per time), that is between t=0 and t=1 12 frames are drawn
- To calculate linear interpolation, we have just to calculate the difference between each frame

```
(180 - 0) / (10 * 12 - 0 * 12) = 180 / 120 = 1.5°
```

- This means that at every frame, our bone have to rotate 1.5° respect the previous frame.

Coding

Well, before starting to code, it's a good idea to prepare all the tools you need to develop. So, to create an animation, we need to have something that export the animation. In our case we can use this same demo to create an animation: since we can actually load a model, and move the bones, we just have to allow the exporting of frames.

Since we have the boneDumpTree function, we just call it every time the user asks, pressing the key 'd' on the keyboard. In this way, you can load a model and create the sequence of keyframe you need for the animation.

I just added few lines to the event handler loop

```
case SDLK_d:
        printf("[FRAME]\n");
        boneDumpTree(root, 1);
        break;
```

Now we have just to code our knowledge.

I'll adopt the bone-related keyframe model, since it have many advantages over the skeleton-related one. Also I'll add some fields in our Bone structure - in addition to the keyframes ones - to keep track of the change between two frames, to avoid more calculations than what's necessary.

So, let's restyle our Bone structure

```
typedef struct
{
        Uint32 time;
        float angle, length;
} Keyframe;

typedef struct _Bone
{
        char name[20];                  /* Remember to prefer numeric IDs */
        float x,
                y,
                a,
                l,
                offA,                           /* Offsets measures for angle and length */
                offL;

        Uint8 flags,
                childCount;

        struct _Bone *child[MAX_CHCOUNT],
                          *parent;

        Uint32 keyframeCount;    /* Number of keyframes */
        Keyframe keyframe[MAX_KFCOUNT]; /* Animation for this bone */
} Bone;
```

The structure is done and ready for animation. Now we have to code

- A function to read an animation file
- A function that animate the model

To save and load animation files, we can modify the boneDumpTree and boneLoadStructure functions:

- boneDumpTree can append to each row, a list of 3 values: time, angle and length, for each keyframe
- boneLoadStructure read each row, as before, but in addition if it finds other info, use them as keyframe

Of course I use this technique because I can create, understand and manage animation files just with ASCII files, but in a game you should write bits instead of strings, and of course create better algorithms to handle save and load.

So, here the modified version of the boneDumpTree

```
void boneDumpTree(Bone *root, Uint8 level)
{
        int i;

        if (!root)
                return;

        for (i = 0; i < level; i++)
                printf("#"); /* We print # to signal the level of this bone. */

        printf(" %4.4f %4.4f %4.4f %4.4f %d %s", root->x, root->y, root->a, root->l, root->flags, root

        /* Now print animation info */
        for (i = 0; i < root->keyframeCount; i++)
                printf(" %d %4.4f %4.4f", root->keyframe[i].time, root->keyframe[i].angle, root->keyfr
        printf("\n");

        /* Recursively call this on my children */
        for (i = 0; i < root->childCount; i++)
                boneDumpTree(root->child[i], level + 1);
}
```

And now let's define the boneLoadStructure function, which load these info.

```
Bone *boneLoadStructure(char *path)
{
        Bone *root,
             *temp;

        FILE *file;

        float x,
              y,
              angle,
              length;

        int unusedChildrenCount,
                depth,
                actualLevel,
                flags,
                count;

        Uint32 time;

        char name[20],
             depthStr[20],
             animBuf[1024],
             buffer[1024],
             *ptr,
             *token;

        Keyframe *k;

        if (!(file = fopen(path, "r")))
        {
                fprintf(stderr, "Can't open file %s for reading\n", path);
                return NULL;
        }

        root = NULL;
        temp = NULL;
        actualLevel = 0;

        while (!feof(file))
        {
                memset(animBuf, 0, 1024);
                fgets(buffer, 1024, file);
                sscanf(buffer, "%s %f %f %f %f %d %d %s %[^\n]", depthStr, &x, &y,
                       &angle, &length, &flags, &unusedChildrenCount, name, animBuf);

                /* Avoid empty strings */
                if (strlen(buffer) < 3)
                        continue;

                /* Calculate the depth */
                depth = strlen(depthStr) - 1;
                if (depth < 0 || depth > MAX_CHCOUNT)
```

```
		{
			fprintf(stderr, "Wrong bone depth (%s)\n", depthStr);
			return NULL;
		}

		for (; actualLevel > depth; actualLevel--)
			temp = temp->parent;

		if (!root && !depth)
		{
			root = boneAddChild(NULL, x, y, angle, length, flags, name);
			temp = root;
		}
		else
			temp = boneAddChild(temp, x, y, angle, length, flags, name);

		/* Now check for animation data */
		if (strlen(animBuf) > 3)
		{
			ptr = animBuf;
			while ((token = strtok(ptr, " ")))
			{
				ptr = NULL;
				time = atoi(token);

				token = strtok(ptr, " ");
				angle = atof(token);

				token = strtok(ptr, " ");
				length = atof(token);

				printf("Read %d %f %f\n", time, angle, length);

				if (temp->keyframeCount >= MAX_KFCOUNT)
				{
					fprintf(stderr, "Can't add more keyframes\n");
					continue;
				}

				k = &(temp->keyframe[temp->keyframeCount]);

				k->time = time;
				k->angle = angle;
				k->length = length;

				temp->keyframeCount++;
			}
		}

		actualLevel++;
	}

	return root;
}
```

Finally, we have to animate the bone. The animation I made is simple:

- A flag determine if I should animate or not
- If I should, check the current time
    - If this is a keyframe, I should calculate the offsets using the interpolation
    - If the actual frame isn't a key, I should just change the actual angle/length
    - If this is the last keyframe, start from beginning
- Draw normally using the x, y, a, l values stored in the bone structure

To do this, we need to add a flag in the code:

```
int animating;

/* Start/end animation */
case SDLK_a:
	animating = !animating;
	break;
```

And then we should add something that handle the current frame state. To update the state we have to visit the entire tree, check for the actual state, calculate the interpolation and update the bone values.

The following function, take the root element and the actual time. Then, it checks if the actual time correspond to any of the keyframes. If it is, calculate the interpolation between frames and animate the bones.

```
void boneAnimate(Bone *root, int time)
{
	int i;

	float ang,
	      len,
	      tim;

	/* Check for keyframes */
	for (i = 0; i < root->keyframeCount; i++)
		if (root->keyframe[i].time == time)
		{
			/* Find the index for the interpolation */
			if (i != root->keyframeCount - 1)
			{
				tim = root->keyframe[i + 1].time - root->keyframe[i].time;
				ang = root->keyframe[i + 1].angle - root->keyframe[i].angle;
				len = root->keyframe[i + 1].length - root->keyframe[i].length;

				root->offA = ang / tim;
				root->offL = len / tim;
			}
			else
			{
				root->offA = 0;
				root->offL = 0;
			}
		}

	/* Change animation */
	root->a += root->offA;
	root->l += root->offL;

	/* Call on other bones */
	for (i = 0; i < root->childCount; i++)
		boneAnimate(root->child[i], time);
}
```

To call this function in the main i added

```
if (animating)
	boneAnimate(root, frameNum++);
```

Note that the frameNum counter is incremented only if the animation is active. This allow us to stop the animation and restart when we stopped.

Simple polygon skinning

So we are finally to the skinning section... probably you got tired with working only with bones, and you may see some substance :D

As I said at the beginning, one of the advantages of the skeletal animation is that you have a skeleton which can move, and you can apply a mesh on it to make the mesh follow it's movement. This is great - for example - if you intend to subdivide the development of the game through a team of people: programmers program how the game interact with the bones, while graphics can draw characters meshes.

Of course skinning is great also for modding: in Quake3, you can create your own mesh, put it in the game, and the game will manage it exactly as all the others. Let's start with skinning!

How does a skin work?

To begin, let's say that a skin is nothing more than a mesh. This mesh is composed by weighted vertexes: each vertex is associated to a bone, and can be influenced more or less by the movement of this bone. Moving a bone, results in a movement of the associated vertexes.

So each vertex don't have only coordinates, but also information about a bone and a weight relative to that bone.

Usually you may want associate a single mesh to a bone system, but it's also possible to associate a mesh to each bone. Since this is easiest, we start from this: each bone has a polygon associated with it: moving the bone results in moving the polygon.

When we'll get more experienced, we'll try to associate a whole mesh to the bones by weighting it's vertexes.

Skinning with multiple polygons

To skin a polygon on a bone you can think that each vertex of the bone is just an offset of each vertex from the bone. The drawing is quite simple: you know from the code above that to draw a bone(x, y, a, l) you have just to

1. translate the matrix where the bone starts
2. rotate the matrix of a number of degrees
3. draw the length of the bone
4. translate again the matrix to the ending position

To draw a polygon, you should only to draw the polygon instead of the bone.

Of course the polygon must be designed to fit on the bone when the angle value is 0: if you draw a mesh rotated of 0° on a bone of 30°, the result is wrong, and you see the mesh rotated 30° more than the bone.

To be more clear: the bones I used above, start always in an horizontal position, since I set length to be the translation on the x axis. So, a mesh drawn for these bones must be drawn in an horizontal position.

I did a simple example of skinning a polygon (GL_QUAD) on a line, take a look to the code: simple skinning example

Now we should apply this example to our model, keeping in mind that our system have a hierarchal structure. First we have to find where we can save mesh info: since we attach some vertices to a bone, we can also add a vertex list in the bone, so we have just to load a mesh for each bone and then we can draw the mesh with a little modification of the draw code. So I just defined a Vertex structure, a very simple one, and added an array of 4 elements in my Bone struct:

```
#define MAX_VXCOUNT                             4

typedef struct
{
        float x,        /* Coords */
              y,
              r,        /* Colors or texture infos */
              g,
              b;
} Vertex;

typedef struct _Bone
{
        ...

        Uint32 vertexCount;
        Vertex vertex[MAX_VXCOUNT];
} Bone;
```

Then I created a simple function which create some geoms on the bones. Actually these geoms are generated by the code, but you may prefer to import them from a file.

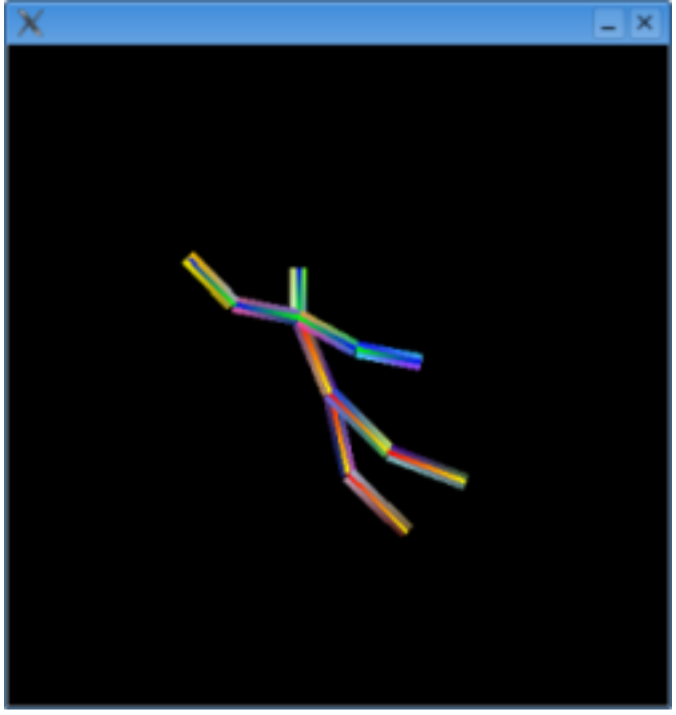
This function is simple: it scan the whole tree and create for each bone an horizontal mesh, with the same len of the bone, giving a random color at each vertex.

```
void boneGenQuads(Bone *root)
{
        int i;

        if (!root)
                return;

        root->vertex[0].x = 0.0;
        root->vertex[0].y = 5.0;

        root->vertex[1].x = 0.0;
        root->vertex[1].y = -5.0;

        root->vertex[2].x = root->l;
        root->vertex[2].y = -5.0;

        root->vertex[3].x = root->l;
        root->vertex[3].y = 5.0;

        for (i = 0; i < 4; i++)
        {
                root->vertex[i].r = (rand() % 256) / 256.0;
                root->vertex[i].g = (rand() % 256) / 256.0;
                root->vertex[i].b = (rand() % 256) / 256.0;
        }

        for (i = 0; i < root->childCount; i++)
                boneGenQuads(root->child[i]);
}
```

And then we've to modify the boneDraw function, to draw the mesh (actually, it draws also the bones over the meshes)

```
void boneDraw(Bone *root, int selected)
{
        ...
        glTranslatef(root->x, root->y, 0.0);
        glRotatef(RAD2DEG(root->a), 0.0, 0.0, 1.0);

        /**** This code draws the quads ****/
        glBegin(GL_QUADS);
                for (i = 0; i < 4; i++)
                {
                        glColor3f(root->vertex[i].r, root->vertex[i].g, root->vertex[i].b);
                        glVertex2f(root->vertex[i].x, root->vertex[i].y);
                }
        glEnd();

        /* Then draw the bones normally */
        glBegin(GL_LINES);
        ...
}
```

Here it is :D



As you can see, the joints are *very* visible, as I said in the introduction. Of course, we can reduce this effect using entire meshes, that we'll see in next section.

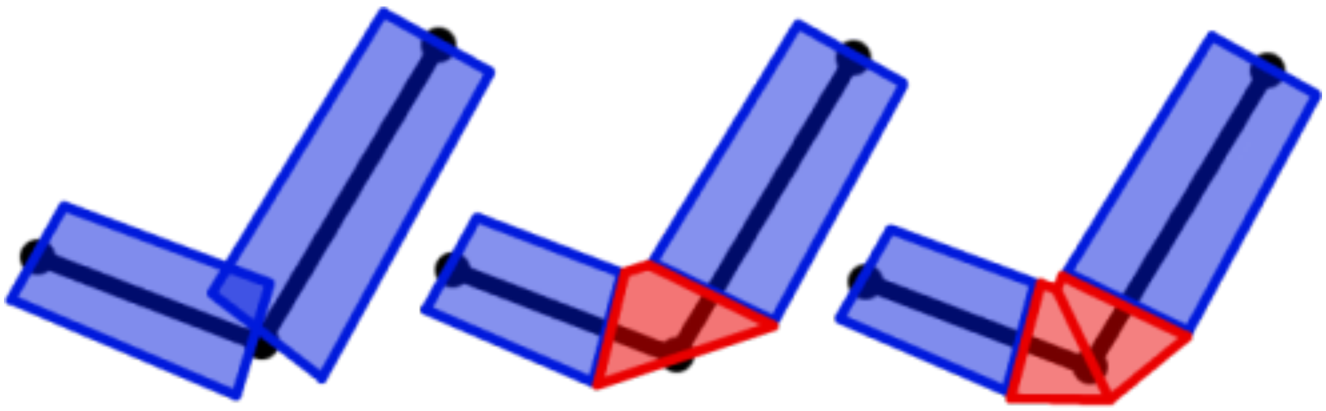
Well, since I made this tutorial just to learn how to use bone structures, and this is exactly what I need, I thin I'll not update this tutorial any more. You have to continue by yourself, learning all the 3D math and complex stuff alone.

Joking ;) Anyway, I reached my objective, since at this point, this kind of structure is usable in simple game.

Working on joints

As said before, having a poly for each bone is simple to implement, but actually it isn't eyecandy since joints are very visible. It would be great to have something that can hide the joints, or at least make them beauty.

We can obtain a better result by connecting two bones like this:



To be honest, I found difficult to implement a such technique, because the bone system we are using is stored in a hierarchal way, so the transformations on the vertices are cumulative and called in a recursive way.

At the moment, in my mind I've 3 solutions

Draw the joint from the parent
: We take all the vertex of the joint, translate them to get their final position, and then create an algorithm that create the joint between these.

Draw the joint from both bones
: When we draw the parent, we draw half bone toward it's child bones, then when drawing the children, we draw the other half toward the father.

Draw the joint from the outside

    We draw all the structure without joints, but while drawing we store data about joint in a global structure. This will be used by a routine to create all the joints at once.

These are the solution I found by myself, I didn't found any paper about this, so I'll write only my experience. If someone know more, please add his knowledge.

The first method seems good to me, but this involve transformations on the vertex: if I'm the parent, with my own translation matrix, and get vertex of my child, I must remember that the vertex is relative to my child, which is relative to me. So I need to translate these vertexes to get their correct position.

The second method is also good, because this is allow to create multiple joints without needing to translate the vertexes, but if the algorithm to create the joint is wrong, this can create inconsistencies in the final result.

The third way don't seem too efficient to me, but this allow you to have a global view of the whole model, so this is good if you intend to perform other operations on the vertex of the model.

Also note that I'll use rectangle as skin, and I'm assuming that the first 2 vertexes of the bone are vertex[0] and vertex[1], and the last two are vertex[vertexCount - 2] and vertex[vertexCount - 1]. These joint will be used by these algorithms, so if you want to give other shapes to the bone, keep in mind that a joint is made by these vertexes.

Parent-side drawing

Drawing from the parent mean that when we draw a bone, we check if it have children. If it is the case, get the children's vertex which are involved in the joint, translate them to be relative to the parent position and finally draw the joint between the bones.

We need to

1. Get the vertices involved
2. Transform the vertices
3. Draw the joint

As said before, I'm assuming 2 beginning and ending vertexes with fixed indexes, so first of all let's get that vertices:

```
int boneGetJoints(Bone *b, Vertex *v)
{
        int i ,
                cnt = 0;

        float m[4 * 4],
                  x0,
                  y0;

        if (!b || !v || !(b->childCount))
                return 0;

        /* We are creating a joint between this bone b and its children
         * so get its ending vertexes
         */
        v[0] = b->vertex[b->vertexCount - 2];
        v[1] = b->vertex[b->vertexCount - 1];
        cnt = 2;

        /* Now get first 2 vertex for each children */
        for (i = 0; i < b->childCount; i += 2)
        {
                v[2 + i] = b->child[i]->vertex[0];
                v[3 + i] = b->child[i]->vertex[1];
                cnt += 2;
        }

        return cnt;
}
```

Now we have to translate them, so let's add translations to the vertices:

```
int boneGetJoints(Bone *b, Vertex *v)
{
        int i ,
                cnt = 0;

        float m[4 * 4],
                  x0,
                  y0;

        if (!b || !v || !(b->childCount))
                return 0;

        /* We are creating a joint between this bone b and its children
         * so get its ending vertexes
         */
        v[0] = b->vertex[b->vertexCount - 2];
        v[1] = b->vertex[b->vertexCount - 1];
        cnt = 2;

        glPushMatrix();
        glLoadIdentity();

        /* Translate the vertex for the length of the bone */
        glTranslatef(b->l, 0.0, 0.0);

        /* Now get first 2 vertex for each children */
        for (i = 0; i < b->childCount; i += 2)
        {
                v[2 + i] = b->child[i]->vertex[0];
                v[3 + i] = b->child[i]->vertex[1];
                cnt += 2;

                /* Transform the vertices */
                glPushMatrix();
                glRotatef(RAD2DEG(b->child[i]->a), 0.0, 0.0, 1.0);

                /* Get the current matrix */
                glGetFloatv(GL_MODELVIEW_MATRIX, m);

                /* Translate the vertexes multiplying the vector with the actual matrix */
                x0 = v[2 + i].x;
                y0 = v[2 + i].y;
                v[2 + i].x = x0 * m[0] + y0 * m[4] + m[12];
                v[2 + i].y = x0 * m[1] + y0 * m[5] + m[13];

                x0 = v[3 + i].x;
                y0 = v[3 + i].y;
                v[3 + i].x = x0 * m[0] + y0 * m[4] + m[12];
                v[3 + i].y = x0 * m[1] + y0 * m[5] + m[13];

                glPopMatrix();
        }

        glPopMatrix();

        return cnt;
}
```

Now we can use these vertex in the drawing function, creating a polygon between these vertex:

```
void boneDraw(Bone *root)
{
        int i;
        float *col,
                col1[] = {1.0, 0.0, 0.0},
                col2[] = {1.0, 1.0, 0.0};

        glPushMatrix();

        /* Draw bones and polygons */

        ...

        /* Get joint vertexes */
        Vertex vert[4];
        int count = boneGetJoints(root, vert);

        /* Draw the joint */
        glColor3f(0.0, 0.0, 1.0);
        glBegin(GL_POLYGON);
                for (i = 0; i < count; i++)
                        glVertex2f(vert[i].x, vert[i].y);
        glEnd();

        /* Translate to reach the new starting position */
        ...

        /* Call function on my children */
        ...
        glPopMatrix();
}
```

Skinning an entire mesh

Now you can skin a model using a set of meshes, which can be connected to a bone. But usually this is not what you want: in fact most of the times the artists produce a single mesh and a skeletal structure for it. Then, it's the programmer that have to work on the skeleton to move the associated mesh. In this section we try to associate a weighted set of vertex (the mesh) to a skeleton (animated).

First, let's do an overview of the work: in this case, we have a list of vertexes and a bone structure. The difference from before is that in this case, we have an entire mesh, this means that all the vertexes are positioned relatively to the mesh. So, to get them positioned correctly, we have to compute their new position relative to the connected bones. This is done using matrix calculations.

Let me tell you that we are going toward some complications: infact, to position each vertex, we have to calculate the absolute position for the connected bones, and then translate the vertex according to them. This is a big waste of computation, because we calculate at each refresh the position for the bones several times. To avoid this waste, programmers usually put the transformation matrix for each bone in the bone structure. The transformation matrix is just a float[16] in the case of OpenGL. This occupies some bytes (64 bytes for each bone, if we're using 32bit floats), but this allow us to do less calculations with matrices.

In this example I'm doing all the calculus, only for sake of this tutorial.

Let's begin with defining the mesh: we said that the mesh is a set of vertex, and each vertex is associated to one or more bones and for each bone it have a weight. So, we define a BoneVertex structure, which contains the data about the bones connected:

```
typedef struct
{
        Vertex v;                       /* Info on this vertex: position color etc */
        int boneCount;                  /* Number of bones this vertex is connected to*/
        float weight[MAX_BONECOUNT];    /* Weight for each bone connected */
        Bone *bone[MAX_BONECOUNT];      /* Pointer to connected bones */
} BoneVertex;
```

And then define a Mesh structure, which contains all the vertexes:

```
typedef struct
{
        int vertexCount;                  /* Number of vertexes in this mesh */
        BoneVertex v[MAX_MESHVXCOUNT];  /* Vertices of the mesh */
} Mesh;
```

Now, we want acquire some data (loading the mesh file). So, let's create a function that read - again - a custom-created file format, in ASCII text (easy to handle and write). The file must contain:

- The number of the vertexes of the mesh
- For each vertex
  - It's coordinates and info (i.e. color)
  - Bones connected
  - Weight for bones

An example for this format is this:

```
2
11.0 22.0 Root 1.0 Head 0.0 Back 0.5
44.0 55.0 Root 1.0 Head 0.5 Back 1.0
```

2 is the number of vertexes in the mesh Then there are 2 rows, one for each vertex: the first and second floats are the position of the vertex, then follow a list of BoneName-Weight values.

So, this is the function used for the mesh loading:

```
void meshLoadData(char *file, Mesh *mesh, Bone *root)
{
        int i, j;
        char buffer[256], blist[256], *tok, *str;
        FILE *fd = fopen(file, "r");

        int id;
        float x, y, w;

        /* Get the number of vertexes in this mesh */
        fgets(buffer, 256, fd);
        mesh->vertexCount = atoi(buffer);

        /* Now read the vertex data */
        for (i = 0; i < mesh->vertexCount; i++)
        {
                fgets(buffer, 256, fd);
                sscanf(buffer, "%f %f %[^\n]\n", &x, &y, blist);
                mesh->v[i].v.x = x;
                mesh->v[i].v.y = y;

                str = blist;
                j = 0;
                while ((tok = strtok(str, " ")))
                {
                        str = NULL;
                        mesh->v[i].bone[j] = boneFindByName(root, tok);
                        printf("Vertex %d bone %s", j, tok);
                        tok = strtok(NULL, " ");
                        mesh->v[i].weight[j] = atof(tok);
                        printf(" is weighted %f\n", mesh->v[i].weight[j]);

                        j++;
                }
                /* Count of relations */
                mesh->v[i].boneCount = j;
                printf("This vertex has %d relations\n", j);

        }

        fclose(fd);
}
```

Now we have a mesh loaded, this is related to some bones, and we need that vertices moves with their bones. So, we want to define a function that, given the mesh, draw its vertices relatively the connected bones. To start, we assume a single bone with weight 1.0.

- Get the current rotation center for the vertex
- Rotate the vertex around the center

The center is actually where the bone is (the absolute coords of the bone). Since actually i didn't store the absolute coords into the bone structure, i've to calculate it with matrix transformations. Of course, it's better to put relative and absolute positions in the bone struct to avoid multiple calculation of the same data.

To get the position of the bone i use OpenGL and some matrix math:

1. We start from a leave in the tree
2. Calculate it's parent position recursively
3. Translate the bone to it's relative position to the parent
4. Rotate the bone of it's angle
5. Get the current matrix
6. Get the bone position

To get the matrix there are these functions

```
void getBoneParentMatrix(Bone *b)
{
        if (!b)
                return;

        if (b->prev)
        {
                getBoneParentMatrix(b->prev);
                glTranslatef(b->prev->l, 0.0, 0.0);
        }

        glTranslatef(b->x, b->y, 0.0); /* For a connected stucture, this is usually 0, 0, 0 */
        glRotatef(RAD2DEG(b->a), 0.0, 0.0, 1.0);
}

void getBoneMatrix(Bone *b, float m[16])
{
        float pm[16];

        if (!b)
                return;

        glPushMatrix();
        glLoadIdentity();

        if (b->prev)
        {
                getBoneParentMatrix(b->prev);
                glTranslatef(b->prev->l, 0.0, 0.0);
        }

        /* Now we are at the end of parent's bone
         * rotate for this bone and
         * get the matrix and
         * return
         */
        glRotatef(RAD2DEG(b->a), 0.0, 0.0, 1.0);
        glGetFloatv(GL_MODELVIEW_MATRIX, m);

        glPopMatrix();
}
```

The bone positions (x, y, z) are the 12, 13 and 14th elements of the matrix. We can get them to get the current bone position. Next step is to define our function, which draws the bones.

```
void meshDraw(Mesh *mesh)
{
        int i,
                n;

        float v[MAX_VXCOUNT * MAX_BONECOUNT][2], /* End vertexes */
                  m[16],
                  tmp[4],
                  x, y;

        n = mesh->vertexCount;

        glPointSize(3.0);
        /* Processing loop */
        for (i = 0; i < n; i++)
        {
                glPushMatrix();
                glLoadIdentity();
                /* Get the bone position */
                getBoneMatrix(mesh->v[i].bone[0], m);
                x = m[12];
                y = m[13];

                /* Go to the bone position */
                glTranslatef(x, y, 0.0);

                /* Rotate the vertex relatively the bone position */
                glRotatef(RAD2DEG(getBoneAngle(mesh->v[i].bone[0])), 0.0, 0.0, 1.0);

                /* Get the matrix */
                glGetFloatv(GL_MODELVIEW_MATRIX, m);

                glPopMatrix();

                /* Save the temporary point */
                tmp[0] = 0;
                tmp[1] = mesh->v[i].v.y;
                tmp[2] = 0;
                tmp[3] = 1;

                /* Multiply the matrix for the point and save the vertex */
                v[i][0] = tmp[0] * m[0] + tmp[1] * m[4] + m[12];
                v[i][1] = tmp[0] * m[1] + tmp[1] * m[5] + m[13];
        }

        /* Draw loop */
        glPushAttrib(GL_ALL_ATTRIB_BITS);

        glBegin(GL_POINTS);
        for (i = 0; i < n; i++)
                glVertex2f(v[i][0], v[i][1]);
        glEnd();

        glPopAttrib();
}
```

Here a two-pass drawing is done since you may prefer to handle vertex to build quads, triangles or lines. To do a single-pass drawing, you should remove not-allowed GL functions and use only external functions to do matrix operations (since these aren't allowed between glBegin and glEnd).

Very good, now it's time to weight out vertexes on multiple bones. As we said, a weighted vertex is influenced by one or more bones, and for each bone the movement can be proportional to a constant: the weight. Usually this constant is a real between 0 and 1, and the sum for all the weights is 1.

When we draw a weighted vertex, we have to apply the linear interpolation. The calculus is done with:

$$v' = \sum_{i=0}^{n} w_i M_i v$$

with

- v' is the resulting vector
- v is the non-transformed vector
- n is the number of associated bone
- w_i is weight associated with the bone
- $1 = \sum_{i=0}^{n} w_i$
- M_i is the transformation matrix for the bone

Now we have **only** to modify a little our function. In the previous definition, we assumed that only the first relation was valid. Now we know how to handle multiple bones, so we just have to create a loop that solve that formula.

```
void meshDraw(Mesh *mesh)
{
        int i,
                j,
                n;

        float v[MAX_VXCOUNT * MAX_BONECOUNT][2], /* End vertexes */
                  m[16],
                  tmp[4];

        n = mesh->vertexCount;

        glPointSize(3.0);

        tmp[0] = tmp[1] = 0.0;
        tmp[2] = 1.0;
        tmp[3] = 1.0; /* w is always 1.0 */

        /* Processing loop */
        for (i = 0; i < n; i++)
        {
                v[i][0] = v[i][1] = 0.0;
                tmp[0] = mesh->v[i].v.x;
                tmp[1] = mesh->v[i].v.y;

                /* Loop thru the relations with each bone */
                for (j = 0; j < mesh->v[i].boneCount; j++)
                {
                        glPushMatrix();
                        glLoadIdentity();

                        /* Get the jth bone position */
                        getBoneMatrix(mesh->v[i].bone[j], m);

                        glTranslatef(m[12], m[13], 0.0);
                        glRotatef(RAD2DEG(getBoneAngle(mesh->v[i].bone[j])), 0.0, 0.0, 1.0);

                        glGetFloatv(GL_MODELVIEW_MATRIX, m);
                        glPopMatrix();

                        v[i][0] += (tmp[0] * m[0] + tmp[1] * m[4] + m[12]) * mesh->v[i].weight[j];
                        v[i][1] += (tmp[0] * m[1] + tmp[1] * m[5] + m[13]) * mesh->v[i].weight[j];
                }
        }

        /* Draw loop */
        glPushAttrib(GL_ALL_ATTRIB_BITS);

        glBegin(GL_POINTS);
        for (i = 0; i < n; i++)
                glVertex2f(v[i][0], v[i][1]);
        glEnd();
        glPopAttrib();
  }
```

I created a mesh file, with some weighted vertexes, to apply to the animated humanoid of the previous section, and this works. You can find the finished and cleaned code in the bottom of the page.

Very good, the first coding part is finished! This has been **very** useful to me - and hope to you - to understand how bones works. Of course, there are plenty of things that must be fixed in the code - if you intend to use it in a production game. In the next section we'll talk about it.

A 3D application using bones

We are finally here: we learned something about bones with the previous section, but now we should optimize it and prepare it for the third dimension. As I said, the above code is full of nasty things. In this section we try to get them better, to gain performances and precision with calculus.

A check before starting

To begin, I start saying that it would be a *great* idea to keep a relative and absolute transformation matrices in the bone. This helps a lot, because when we want to refer to a bone's space coordinate, we have just to use the bone relative matrix, and when we want to position the bone in the space, there is the absolute matrix.

We can remove some code - gaining speed, and as you probably noticed, this kind of work requires handling of matrices, so it's better if we can use the OpenGL code to handle them (we assume that OpenGL can do faster matrix operations that our simple functions). If you want to use an external fast math library, do it now.

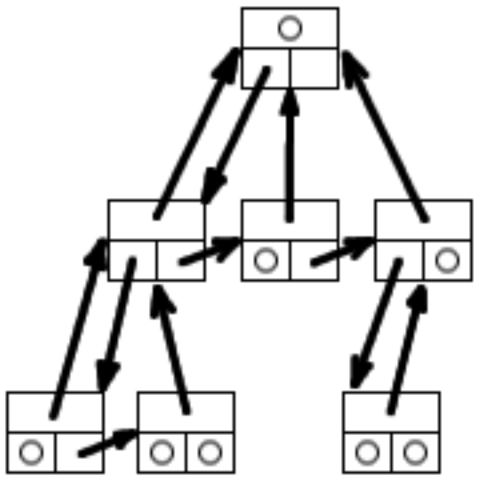
Also I think it's a good idea to decide now the measure of angles: since OpenGL uses degrees for angles, I suggest to use degrees everywhere, and defining two macros for funcions like sin and cos for using degrees.

Data structures

In the 2D section, I created a very simple and intuitive data structure for the bones and for the vertexes, because *there* it was important to understand how bones works. *Here* we want to gain performances, so a different structure is used. As said before, i'll use two matrices to store transformation of the bone in relative and absolute space.

A matrix, in the OpenGL context, is a 16 element array of float or double. I'll use float since it gives a good quality in the calculations and a good speed. If you need more precision in calculations, doubles are for you. Using a matrix we can determine where the bone is located, and which is its rotation angle, but we have to keep it's length as a separate float.

And of course we want to keep the hierarchal structure, but in the previous example I used arrays to keep information about children, now we use a different tree



This tree (that I call s-tree, but i'm quite sure it have a different name :P) allow us to keep an unlimited number of level and children and, in our case, performances are good since in our algorithms we don't need a direct access to the child.

In addition, it's a good idea to avoid strings to refer to bones ;) If you want to use a string for giving a name to a bone, you can do that, but it's faster to give a numerical ID to each bone.

So a structure for these bones can look like this:

```
typedef struct _Bone
{
int id;
float absMatrix[16], relMatrix[16];
float length;
struct _Bone *parent, *child, *brother;
} Bone;
```

Source codes

- Test source for 2D bones - This is a very basic implementation of this tutorial. Improvements will come later...
- File:Bbs skinning.c
- File:Bbs snakeSkel.txt
- File:Bbs snakeMesh.txt

External Links

- Tutorial with Source Code in C++ for GPU skinned skeletal animation using GLSL (http://voxels.blogspot.com/2014/03/skinned-skeletal-animation-tutorial.html)
- Hierarchical Skeletal Data (http://molecularmusings.wordpress.com/2013/02/22/adventures-in-data-oriented-design-part-2-hierarchical-data/)

Retrieved from ‘http://content.gpwiki.org/index.php?title=OpenGL:Tutorials:Basic_Bones_System&oldid=32349’
Categories: C and OpenGL | All C articles | All OpenGL articles
Navigation menu

Navigation

Namespaces

- Page
- Discussion

Variants

Views

- Read
- View source
  View history

.
