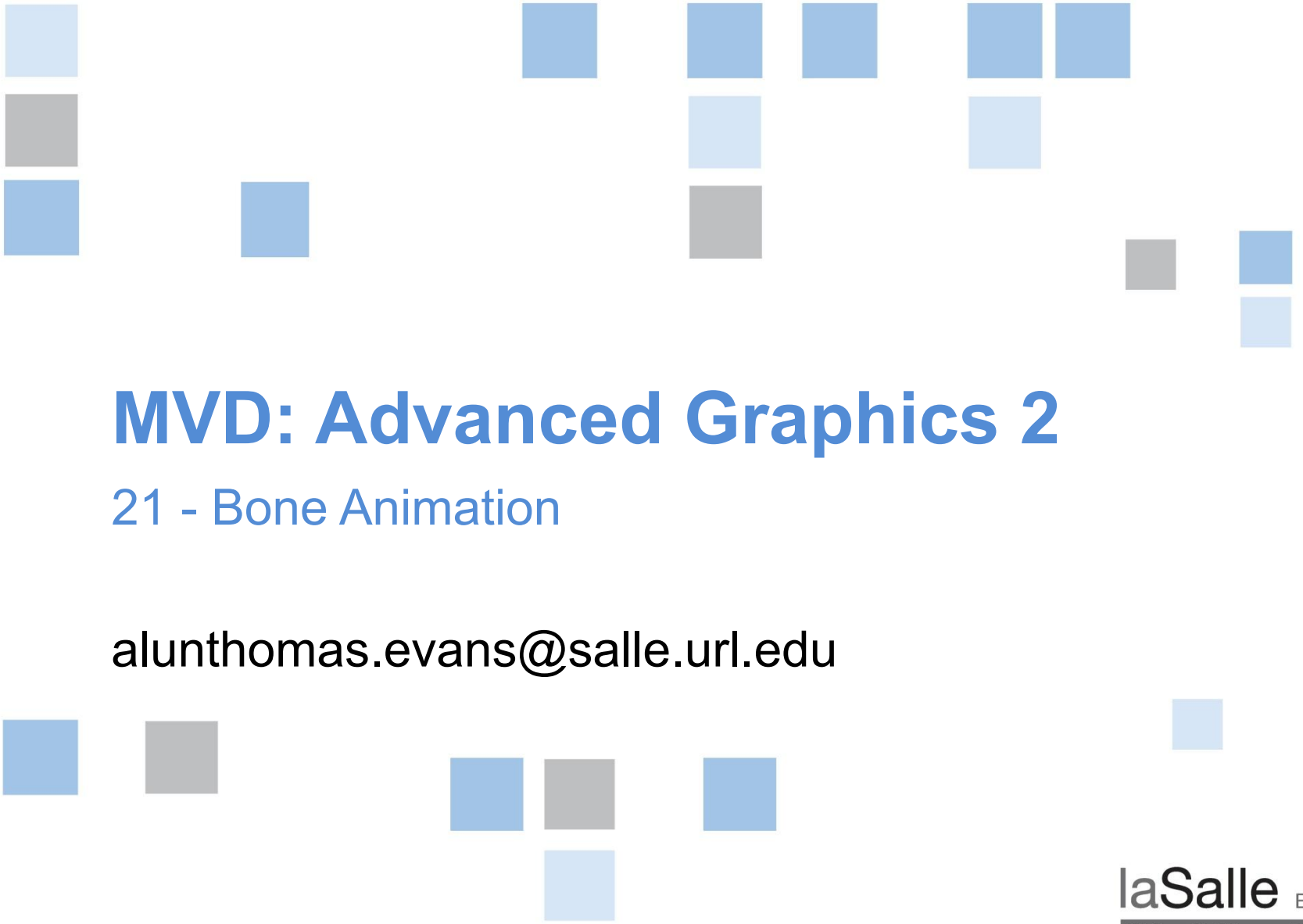


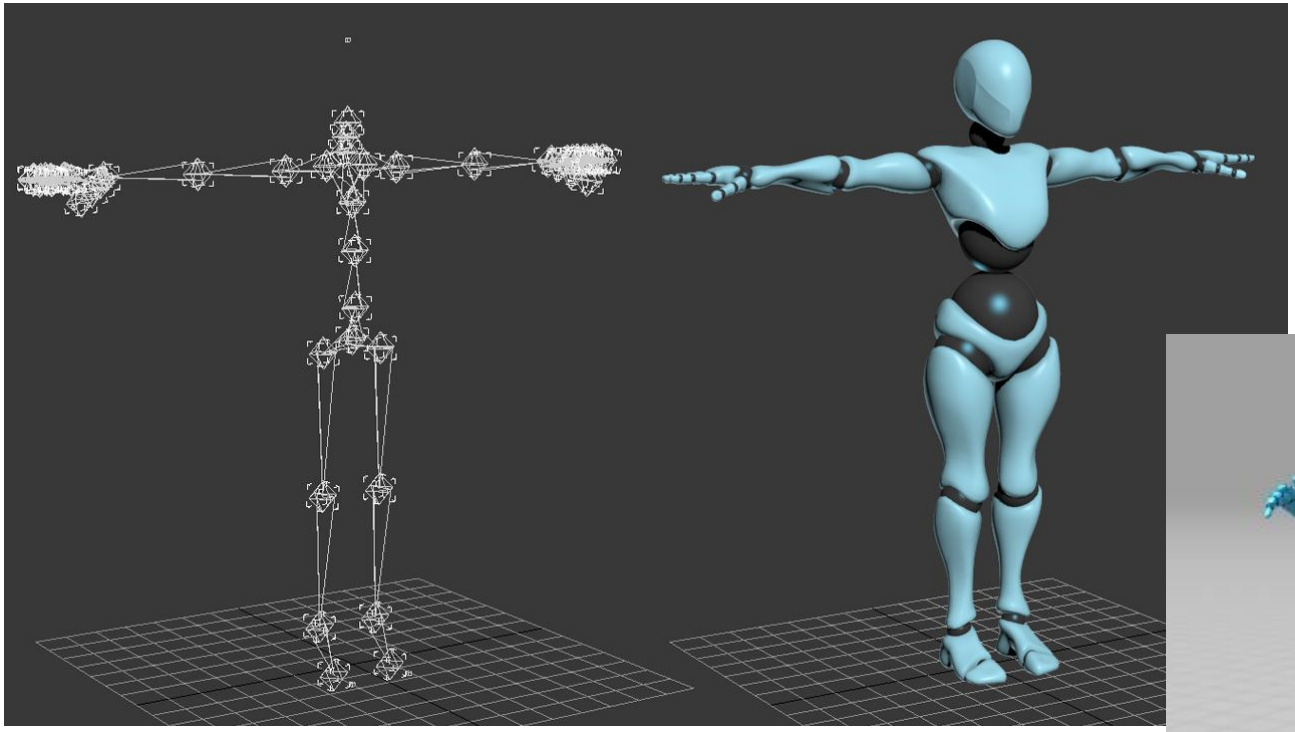
MVD: Advanced Graphics 2

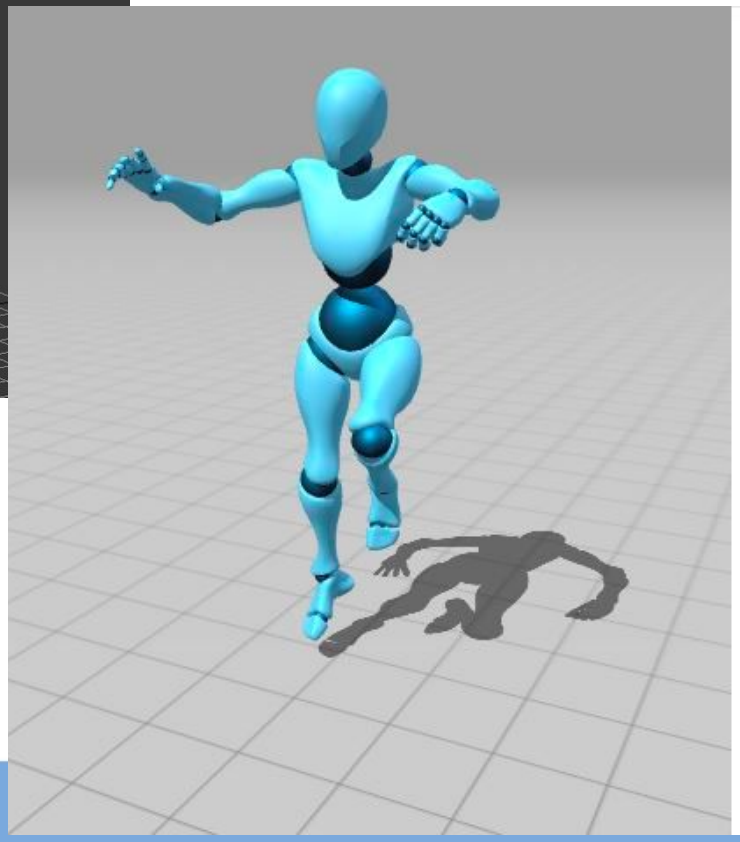
21 - Bone Animation

alunthomas.evans@salle.url.edu

laSalle ENG
Universitat Ramon Llull

Skeletal Animation





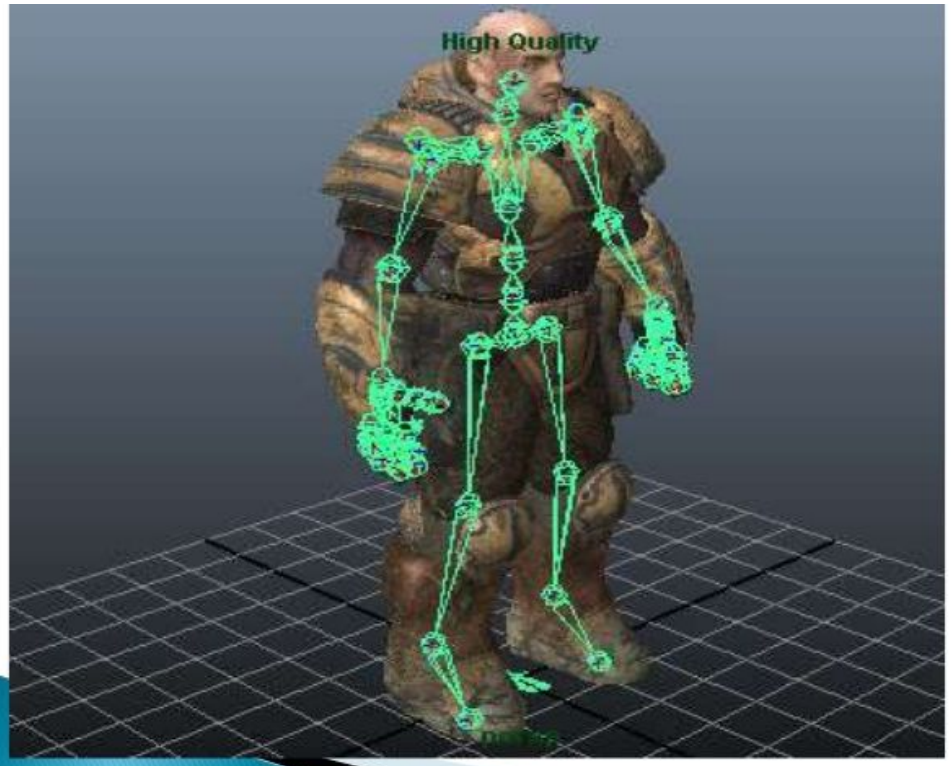
MVD
Alun Evans – alunthomas.evans@salle.url.edu

What is Skeletal Animation

Skeletal Animation is an animation technique where objects are represented by two separate components

1) The Skeleton

2) The ‘Skin’



2D skeletal animation

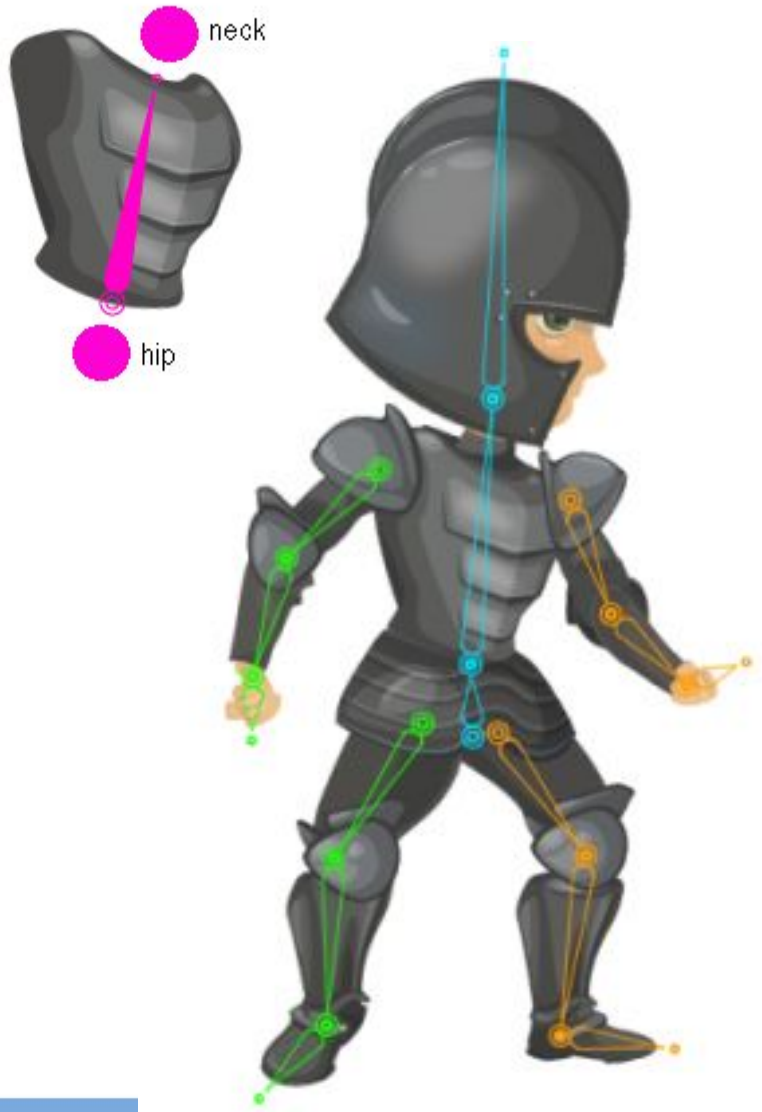
We usually think of skeletal animation as a 3D technique.

But you can also think of it in 2D, if the ‘skin’ is composed of separate images

2D skeletal animation

A series of 2D vectors form the ‘skeleton’. These are called ‘joints’ or ‘bones’.

Each joint is associated with an image (sprite). When you transform the joint, you apply the same transformation to the image

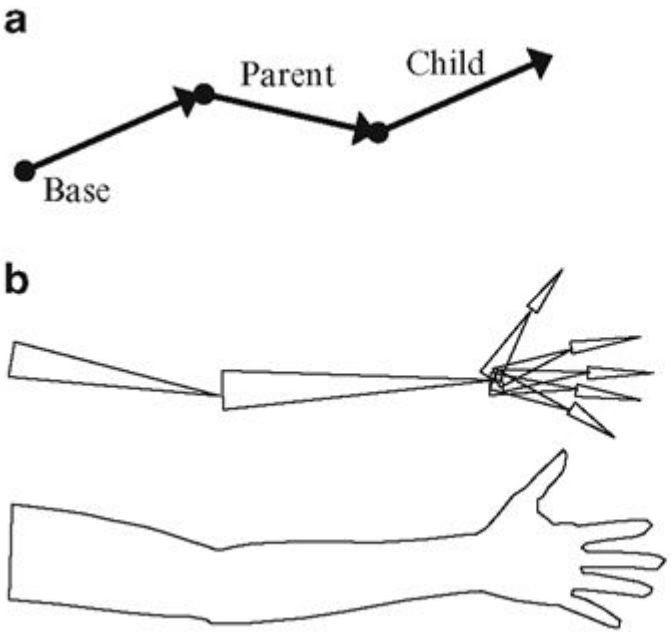


Joint chains

Joints are usually organised into hierarchical trees called ‘Joint Chains’ (or ‘skeletons’; sometimes ‘armature’).

When you move a joint higher up in the chain, all the other joints are moved relatively.

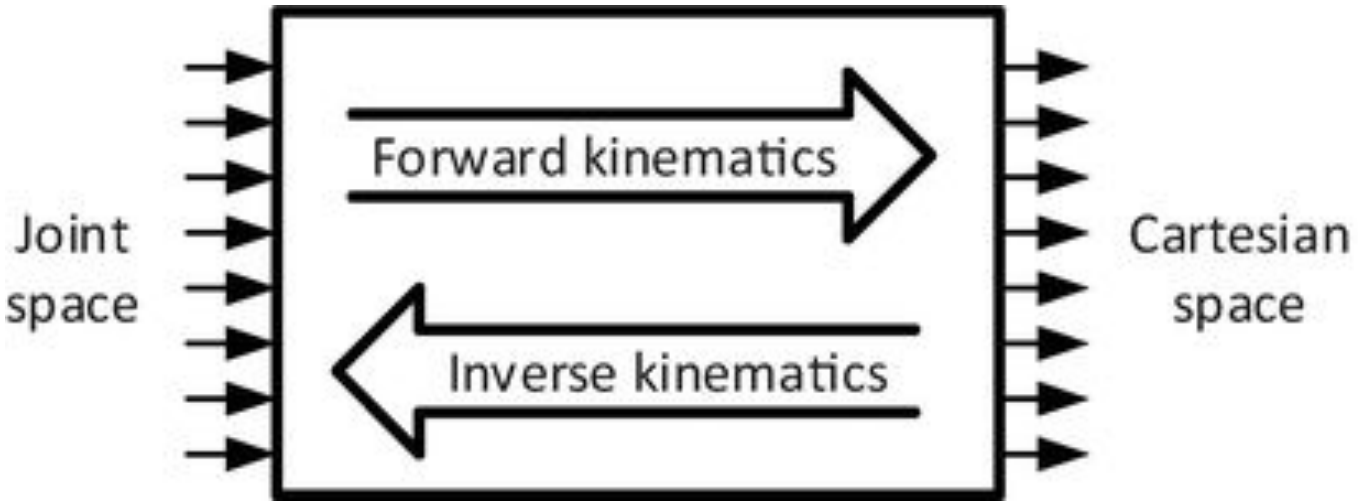
This is called FORWARD KINEMATICS



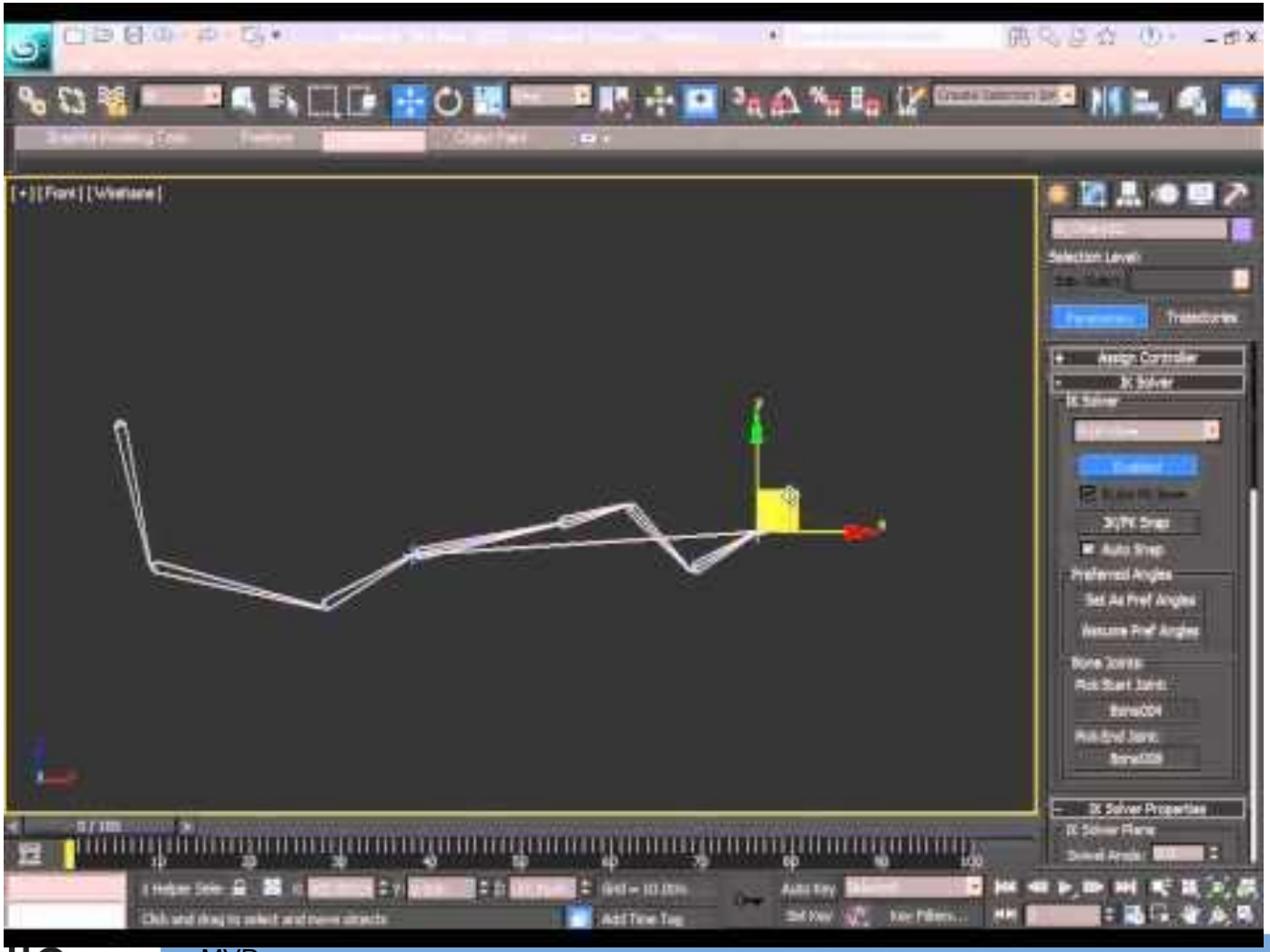
‘Inverse’ Kinematics is more useful for animators

Forward = “move the elbow, what’s the position of the hand?”

Inverse = “move the hand, what’s the position of the elbow?”



Simple IK in 3D Studio Max (skip to 1m)



How IK solvers work

Need to calculate **minimum angle for each joint**.

For short chains (e.g. 3 joints) the minimum angle can be calculated using trigonometry and algebra.

For longer chains, most solvers employ a **gradient descent energy minimising algorithm**, where the energy is a function of the sum of the angles in the chain

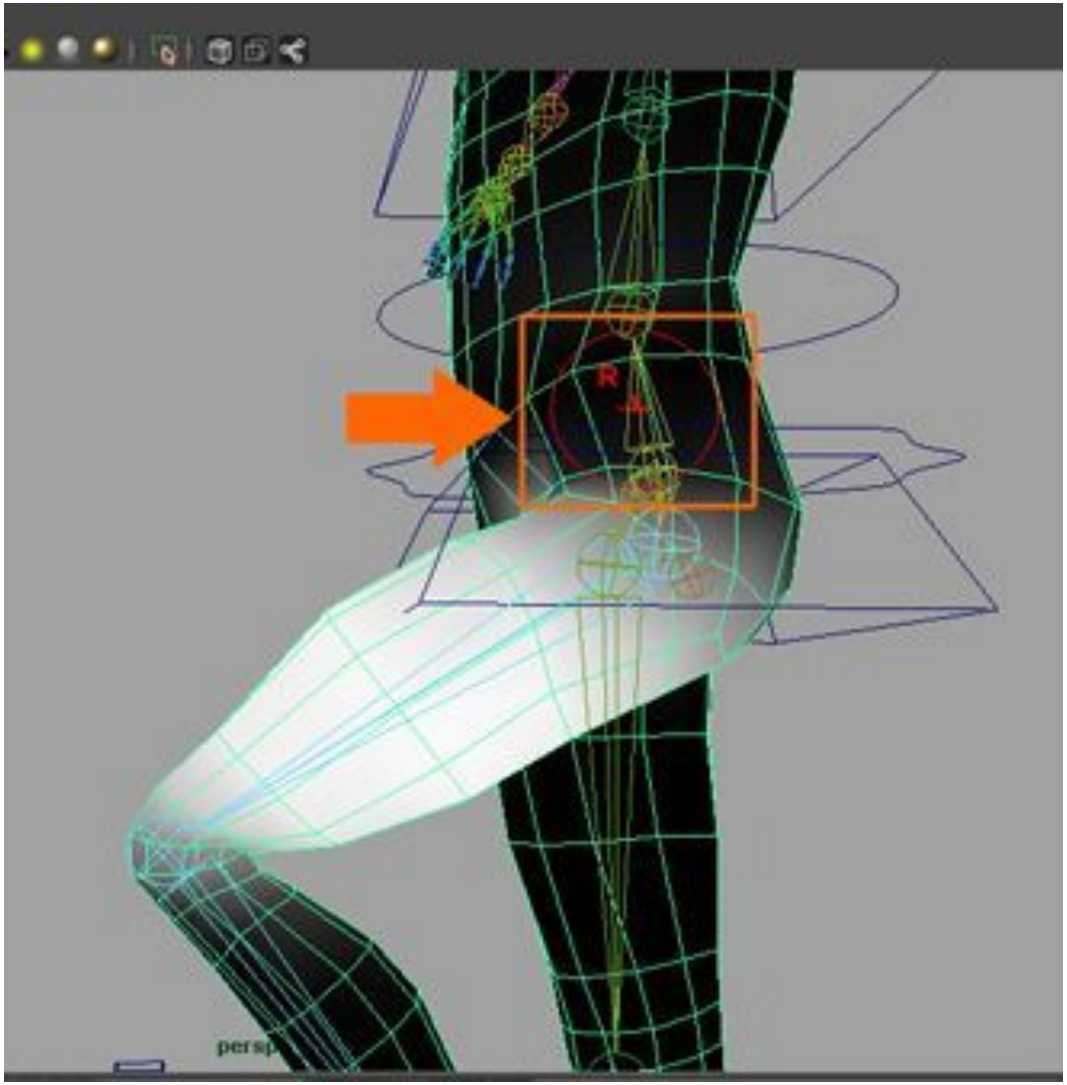
3D mesh - the ‘skin’

In 3D, instead of associating a whole image with a joint, we must associate a set of vertices to each joint.

Each vertex of the mesh is assigned

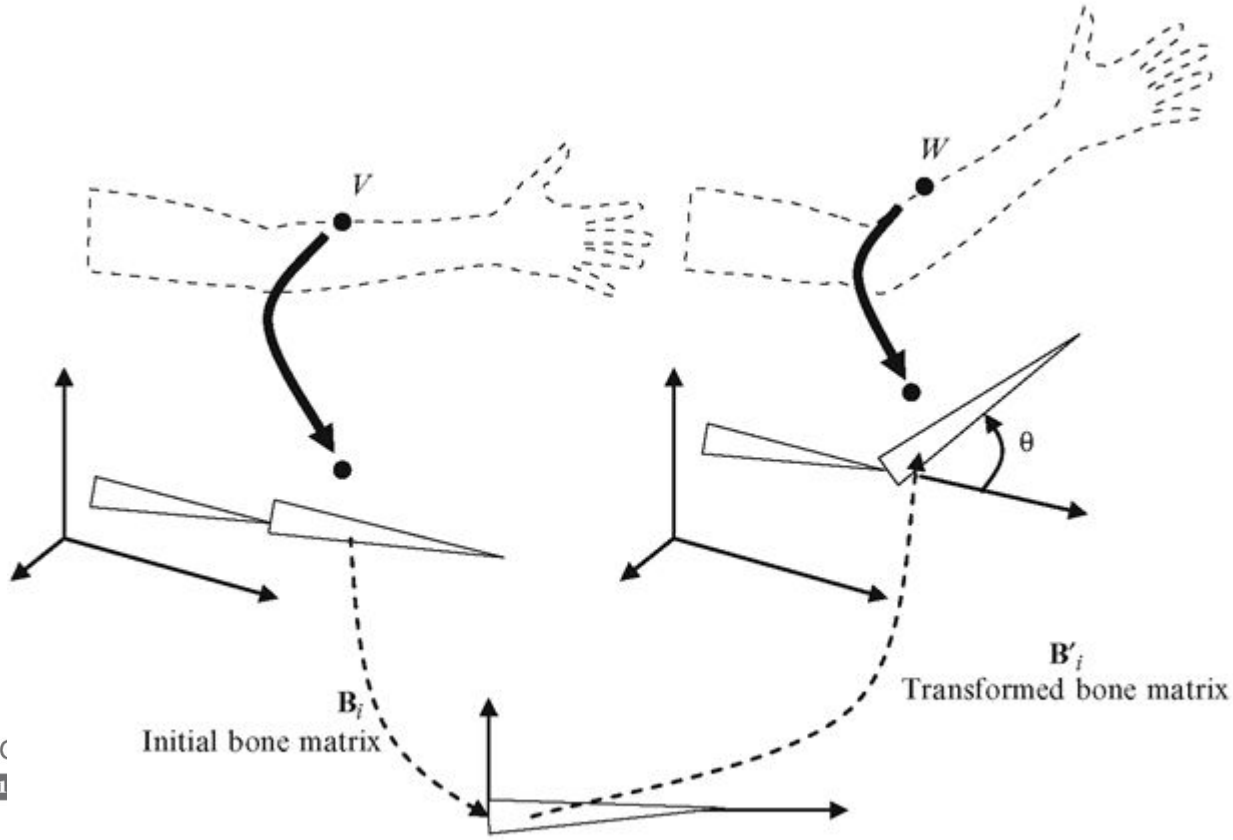
i) a joint index

ii) a weight (0.0->1.0)

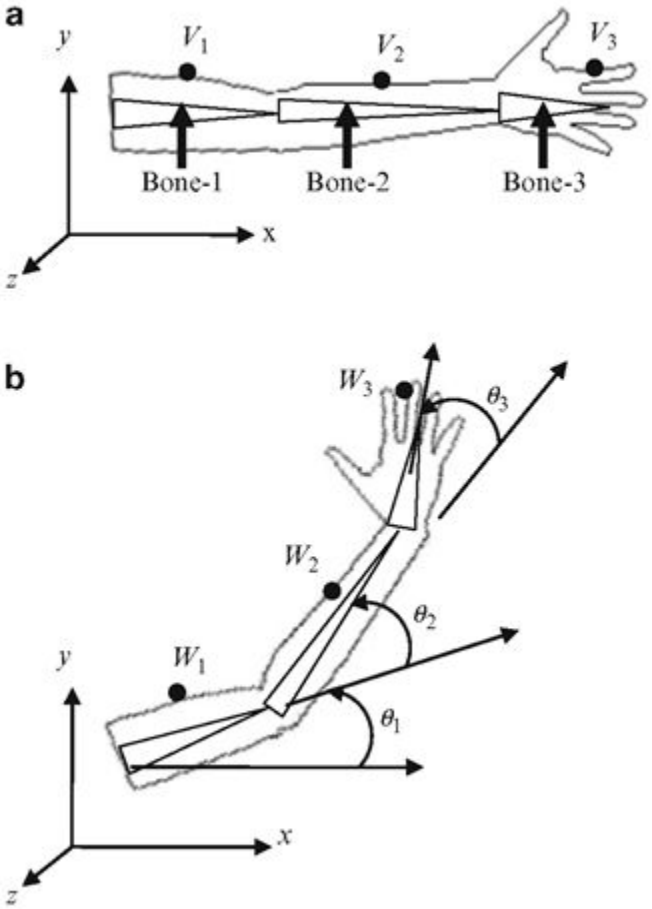


Vertex weights

If a vertex has a weight of 1.0 for joint id *x*, then 100% of *x*'s transformation is applied to that vertex



Joint chain hierarchy applied to vertex



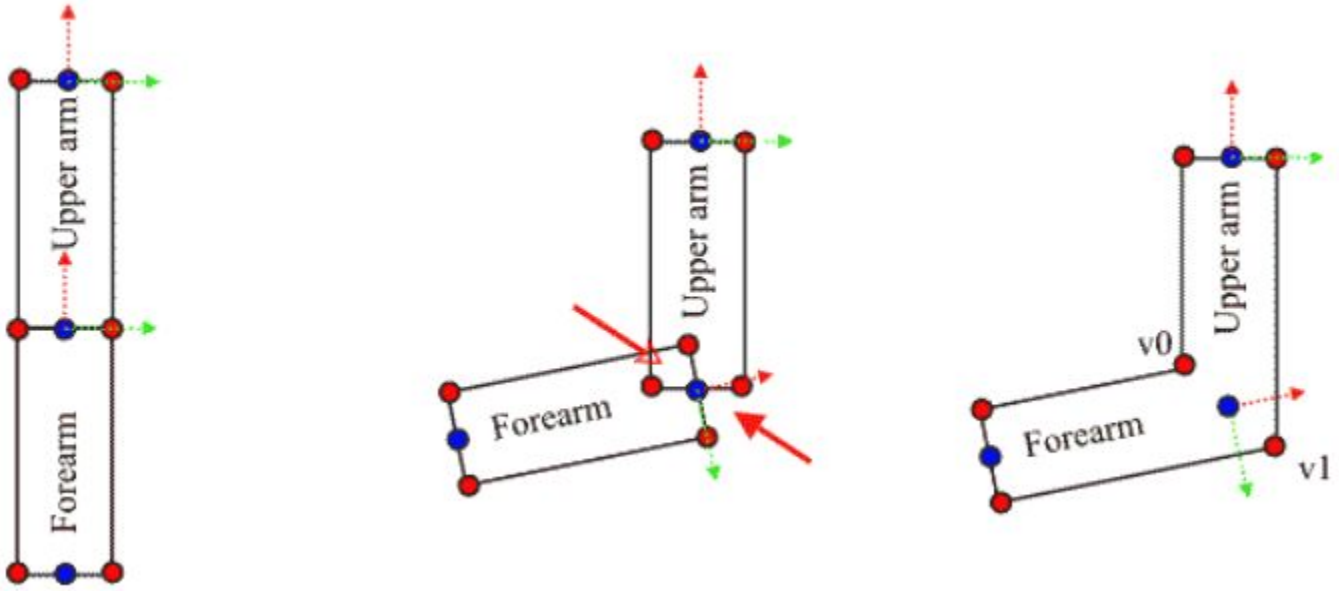
Don't forget that the global transformation matrix for each joint is it's model matrix multiplied by that of it's parent.

In this case vertex V3 is modified to position W3 by:

W3matrix = Bone1 * Bone2 * Bone3

Multiple joint weights

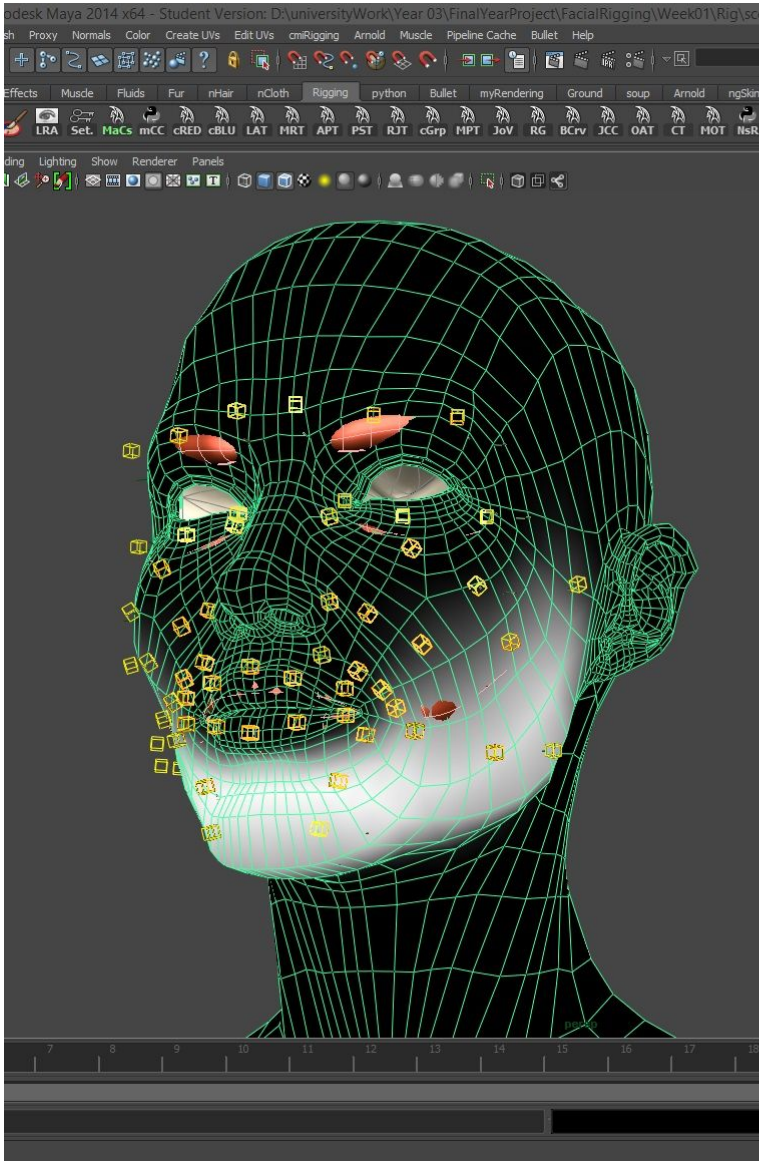
If we associate a vertex with only one joint, we get problems



Multiple joint weights

Vertices can ‘weighted’ by >1 joint (but it’s not mandatory)

Traditionally, the limit is four.



Joint weight table example (for one vertex)

Joint ID	Weight
4	0.5
2	0.1
13	0.2
1	0.3

So every vertex in the mesh has a table like this one.

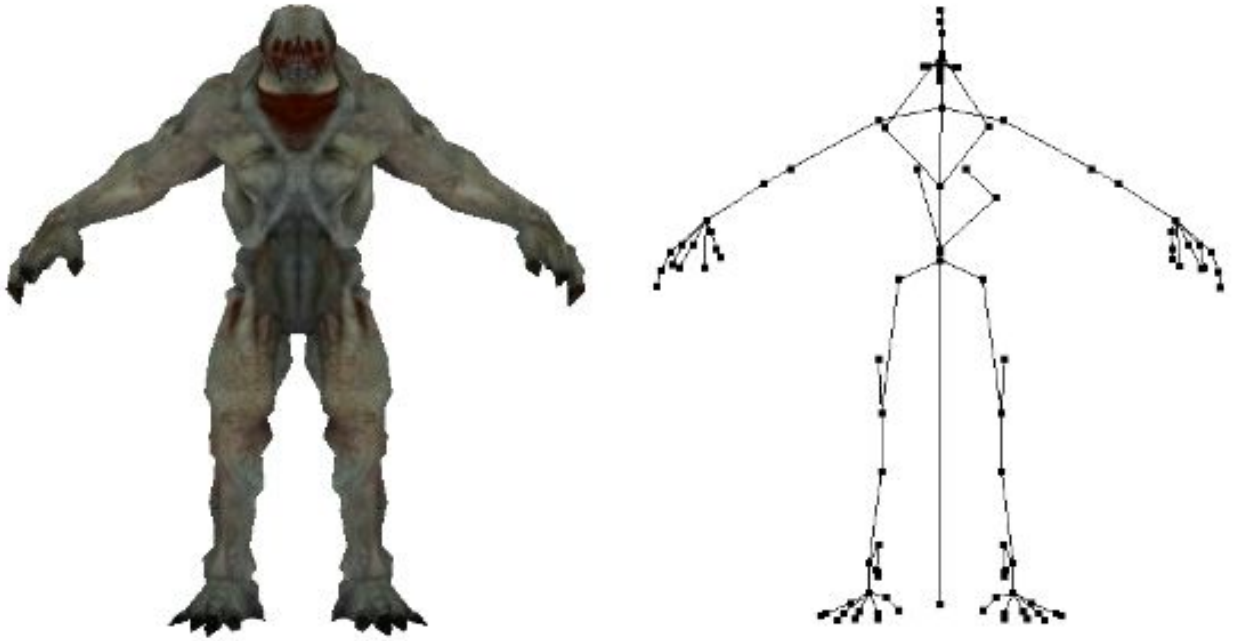
Mathematically the relationship is:

$$position = \sum (t \cdot v) \cdot w$$

The final vertex position is formed by summing the results of multiplying each joint transform t *by either the vertex or weight position* v*, scaled by multiplying by joint weight value* w

Rigging

The process of assigning weights is to each vertex is called rigging. Rigging is carried out with the mesh and joint chain configured into '**bind-pose**' (a position which attempts to expose all vertices of mesh).



Animating

Once a mesh is rigged, it can now be animated.

Animation uses **keyframes**, the same as rigid animation. Each joint has a transformation matrix associated with each frame.

The difference is this:

the matrix for each joint is the transformation

from bind-pose

Skinned Mesh multiplication order

Raw Vertex Position

Skin Global Bind Matrix
(optional ‘skin model matrix’)

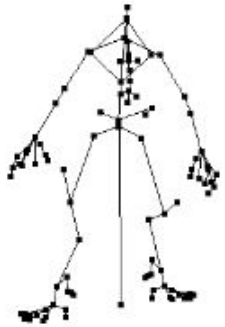
For each joint:
(that affects this vertex)

Joint Bind Matrix
(transform to bind-pose)

Joint transform
(bind-pose to current keyframe)

Joint weight
(for this vertex)

Final Vertex Position



Normals

Do the same but turn 4x4 matrix into 3x3 matrix

(and assume no non-uniform scaling)

Exporting / Importing Animation

Exporting/Importing Skeletal Animation

Need to export/import:

Joints:

- name/id
- parent
- default model matrix
- bind matrix

Skinned Mesh:

- Positions
- Normals
- UVs
- JointID/Weight table
- Global bind matrix

Materials

Lights

Cameras etc.

Animation

- Keyframes
- Matrices for joints

Popular formats

Collada (.dae) - XML based format for scene transfer, fairly logical, plenty of documentation available

http://www.opencollada.org/ - open source exporters for Maya, Max and others, work very well.

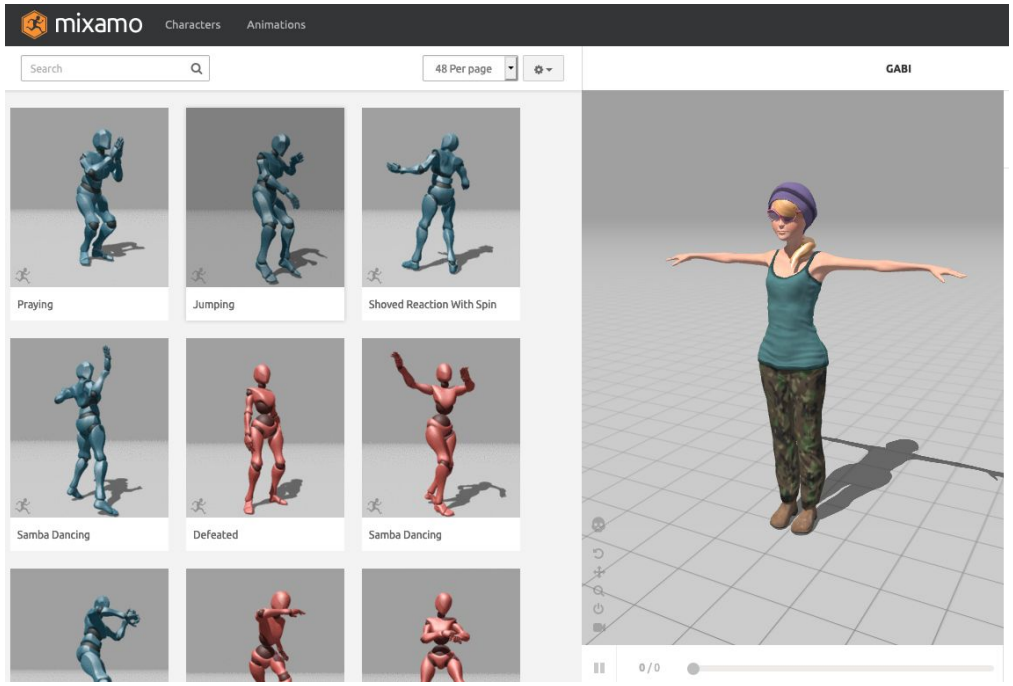
Filmbox (.fbx) - proprietary binary format owned by Autodesk, but also well documented. Autodesk also have released a C++ SDK for parsing the format.

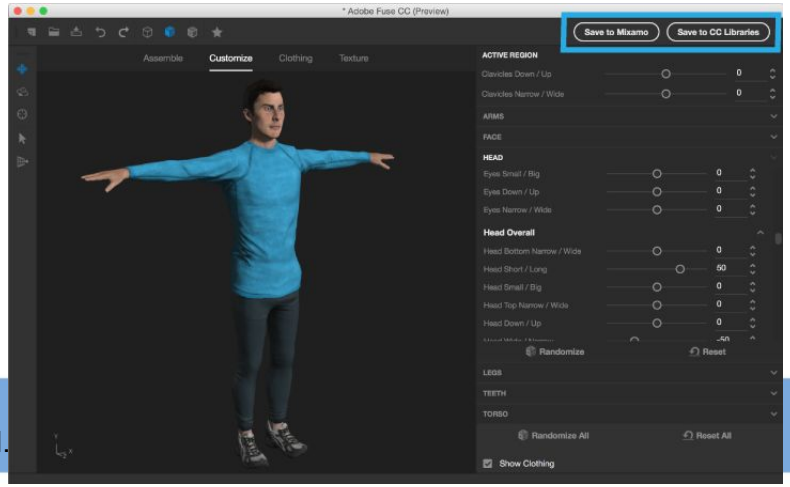
Where to get test animation data?

Mixamo.com - huge library of publicly (for now!) available animations.

Can download in FBX or Collada

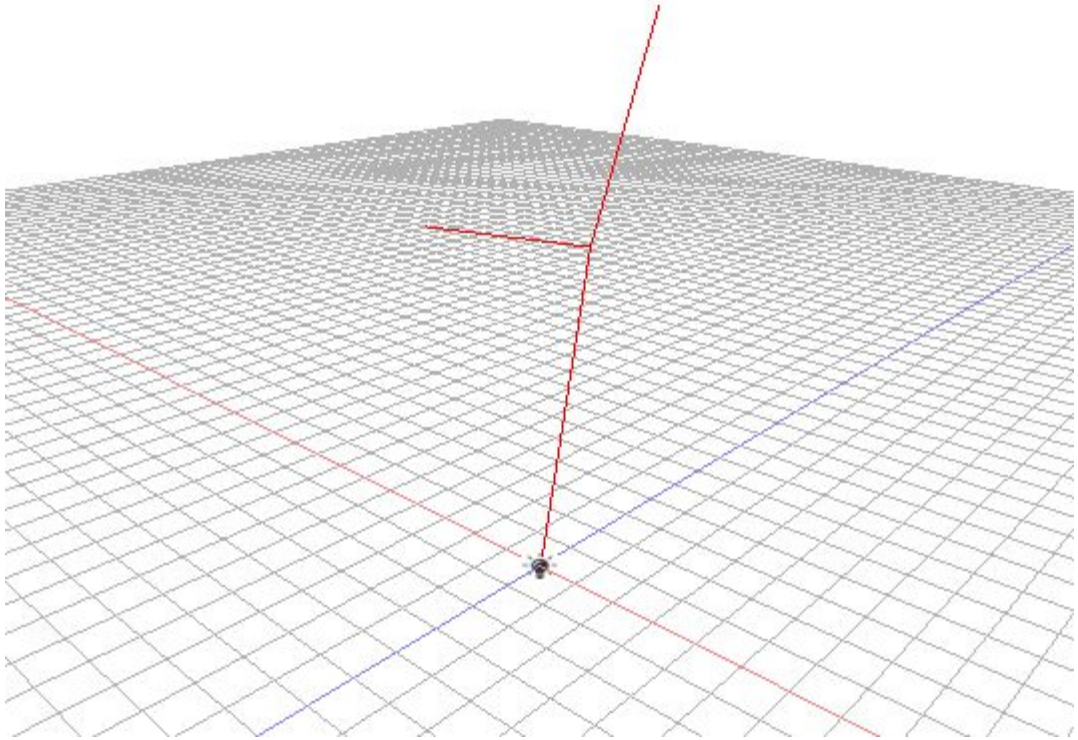
Seamless integration with Adobe Fuse





Today and next week

Start with joint chain



next week, add skin

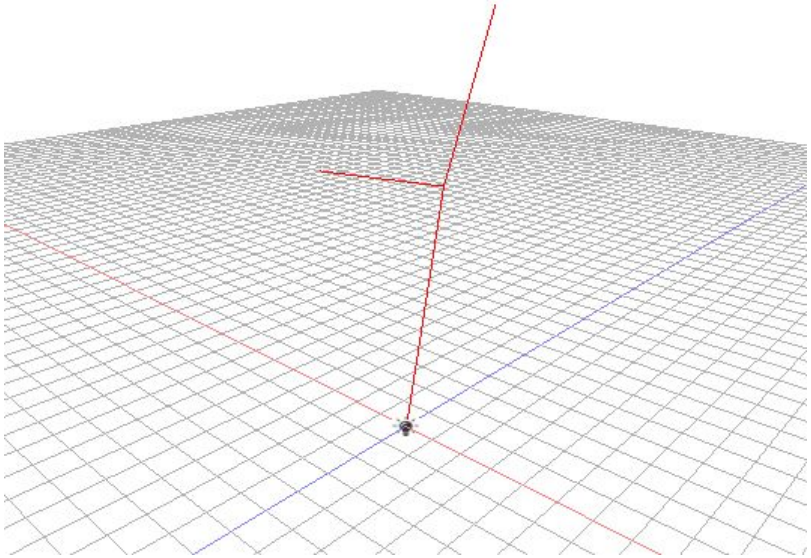
Today's task

I give you:

- a simple Collada parser
- basic Joint and JointChain structures

You have to:

- write the code that draws animates the joint chain



Collada example

Collada is an XML-based format. Parser uses tinyxml lib

```
<library_visual_scenes>
  <visual_scene id="VisualSceneNode" name="tick_anim4">
    <node id="_joint1" name="joint1" sid="joint1" type="JOINT">
      <matrix sid="transform">-0.6940169 0 0.7199587 -0.02479339 0.7199587 0 0.6940169 -0.04958678 0 1 0 0
      0 0 0 1</matrix>
      <node id="_joint1_joint2" name="joint2" sid="joint2" type="JOINT">
        <matrix sid="transform">0.9267039 0 -0.3757923 7.012629 0 1 0 0 0.3757923 0 0.9267039 0 0 0 0 1</
        matrix>
        <node id="_joint1_joint2_joint3" name="joint3" sid="joint2_joint3" type="JOINT">
          <matrix sid="transform">-0.3713907 0.9284767 0 5.340659 0 0 1 0 0.9284767 0.3713907 0 0 0 0 0 1</
          matrix>
          <extra>
            <technique profile="OpenCOLLADAMaya">
              <originalMayaNodeId sid="originalMayaNodeId" type="string">joint3</originalMayaNodeId>
            </technique>
          </extra>
        </node>
        <extra>
          <technique profile="OpenCOLLADAMaya">
            <originalMayaNodeId sid="originalMayaNodeId" type="string">joint2</originalMayaNodeId>
            <visibility sid="visibility" type="bool">1</visibility>
          </technique>
        </extra>
      </node>
```

Joint Chains

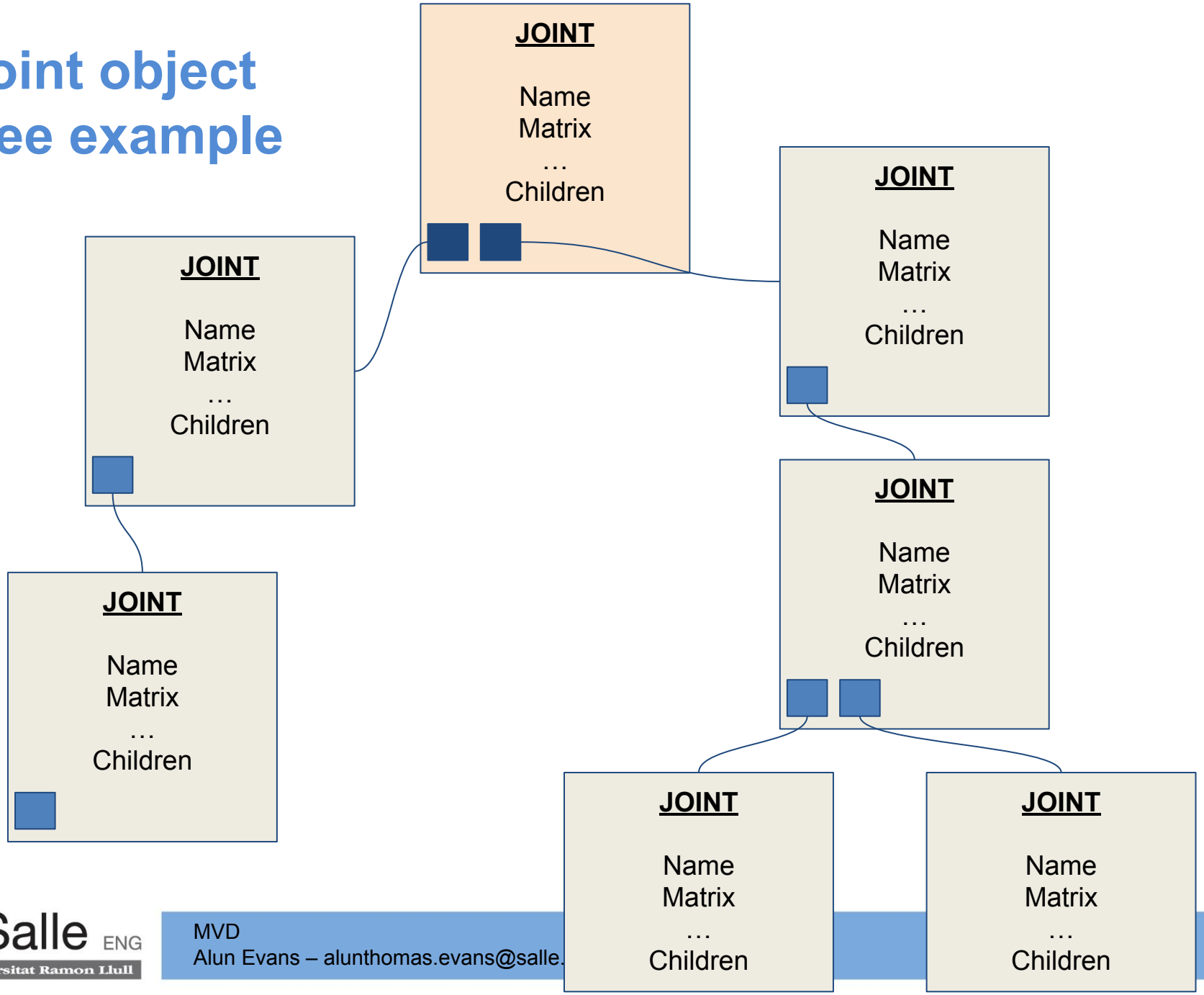
For now, we will make a component called Joint Chain

(we will substitute it next week)

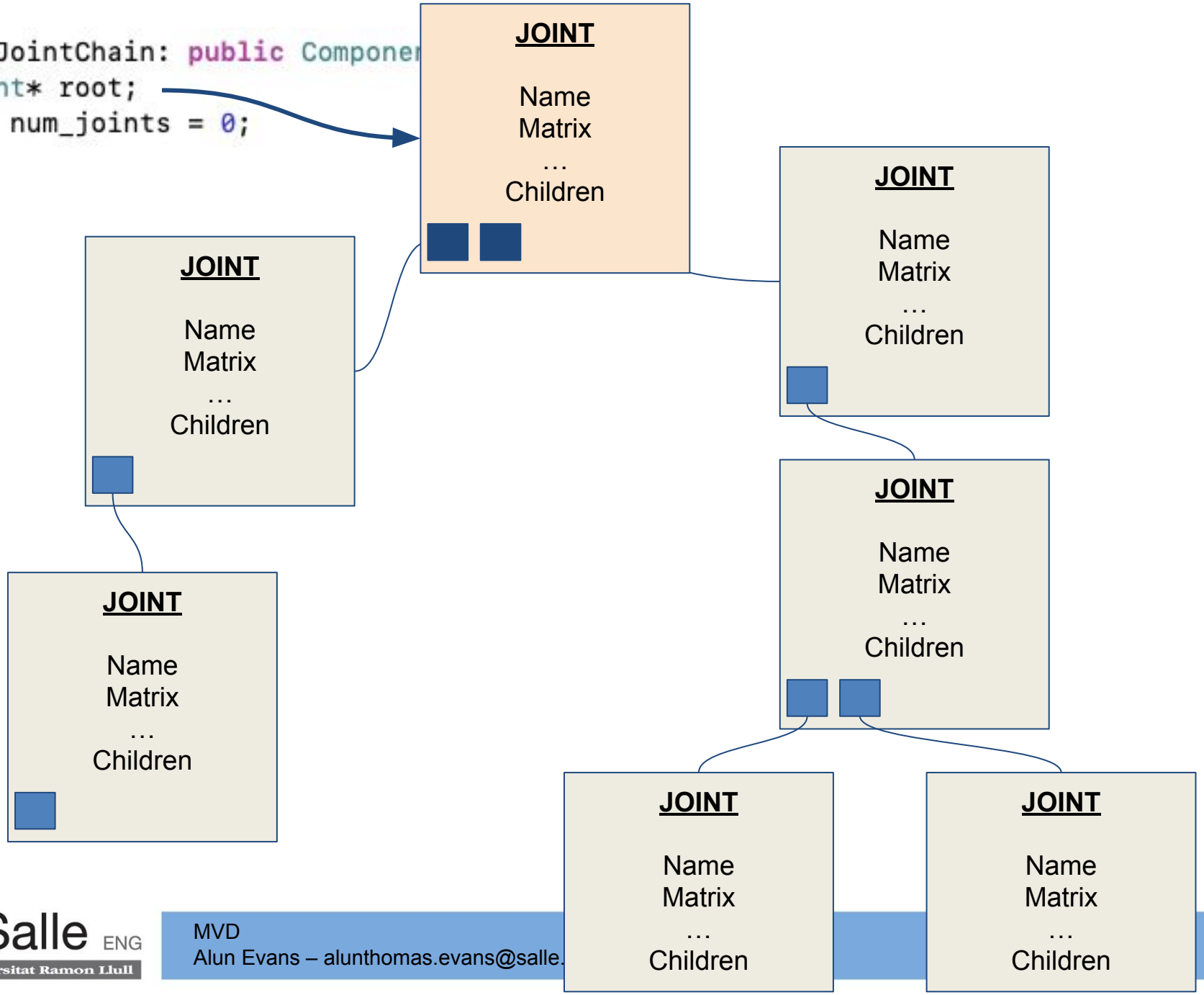
```
struct JointChain: public Component {
    Joint* root;
    int num_joints = 0;
};
```

The ‘chain’ is a Tree structure where each joint represents a position

Joint object tree example



```
struct JointChain: public Componer
    Joint* root;
    int num_joints = 0;
};
```

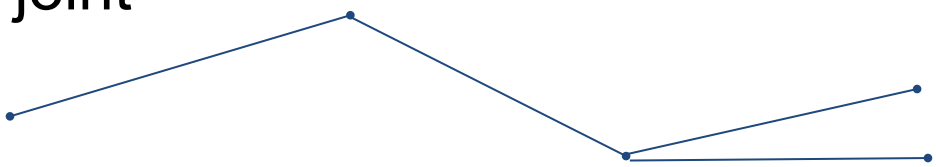


Matrix of joint

Is a transform. But it's also a vector from 'origin'

But in visual representation, is a point, displayced from 'origin'.

To show joint, draw a line from parent joint (child's 'origin') to child joint



Joint struct

Last week we had an ‘Animation’ component

```
struct Animation : public Component {
    std::string name = "";
    GLint target_transform = -1;
    GLuint num_frames = 0;
    GLuint curr_frame = 0;
    float ms_frame = 0;
    float ms_counter = 0;
    std::vector<lm::mat4> keyframes;
};
```

What do we need to make a ‘Joint’?

Joint struct (not a component)

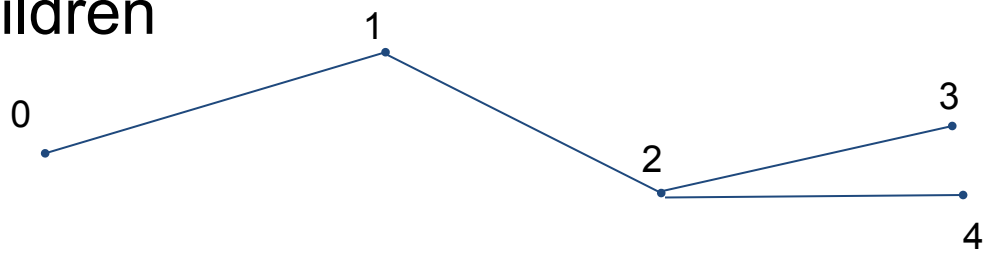
JOINT

```
//core stuff
string Name
string ID
mat4 Matrix

//chain stuff
int index_in_chain
Joint* parent
vector<Joint*> Children

//animation stuff
vector<mat4> keyframes
int num_keyframes
int curr_keyframe
```

For simplicity, let's assume a global fps for now

Collada parser creates joint chain

- sets matrices
- sets parents and children
- creates chain id

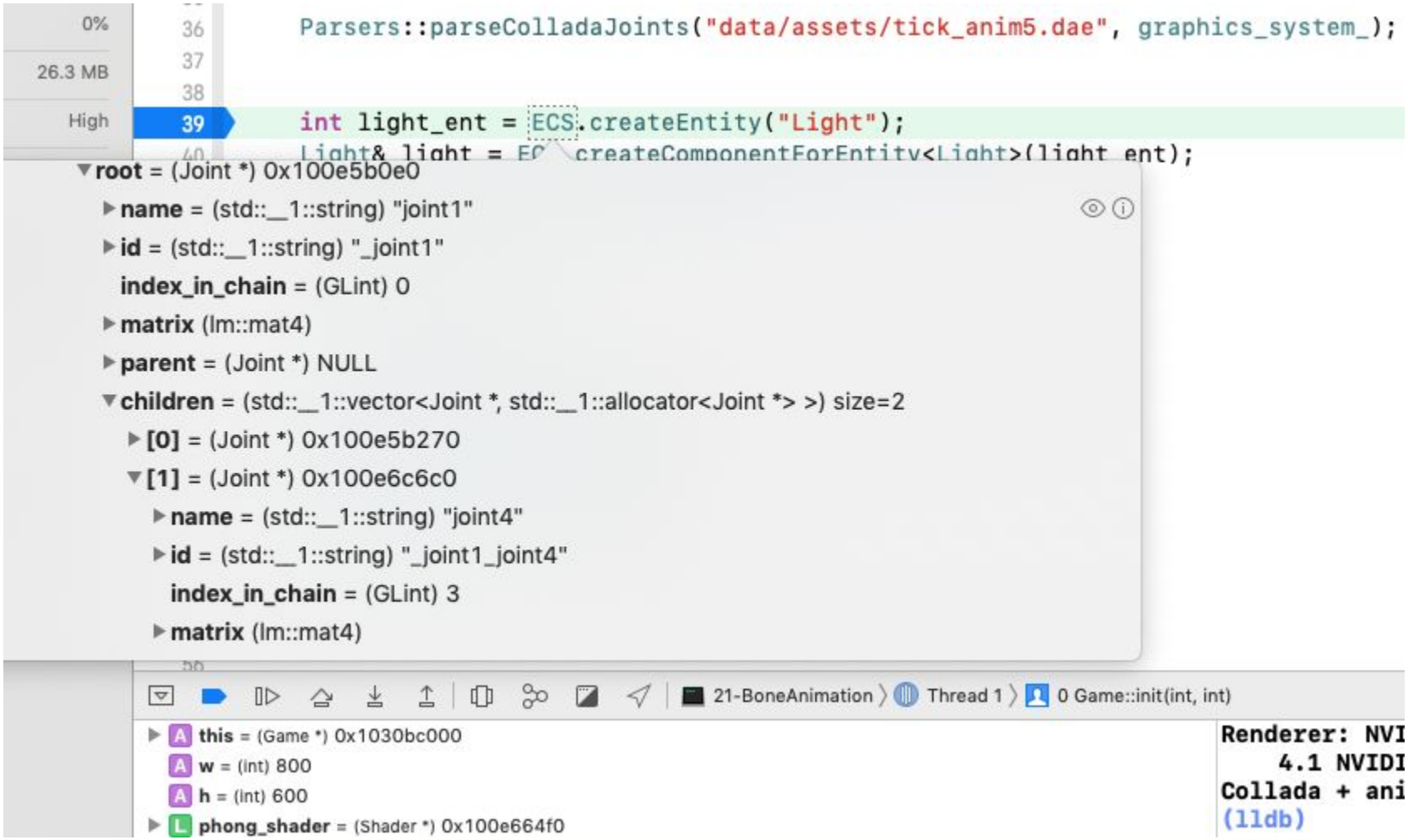


- sets keyframes

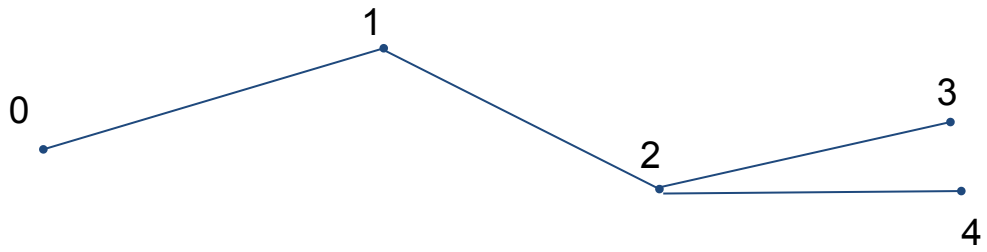
```
//function receives a std::vector of floats which correspond to model matrices of all
//keyframes of animation
//obviously MUST be a factor or 16!
void setKeyFrames(std::vector<float>& raw_floats) {
```

- creates Joint Chain and sets num joints

Test it with the debugger!



Drawing the Joint Chain



We have

- a tree of joints
- each with a model matrix, representing a point

How do we draw it?

Option A

Create an array of vertices.

Set vertex position according to model matrix of joint

Upload to GPU

Draw lines

Option A

Create an array of vertices.

Set vertex position according to model matrix of joint

Upload to GPU

Draw lines

WHY NOT?

Option B

Create an array of vertices, **all at position (0,0,0)**.

~~Set vertex position according to model matrix of joint~~

Upload to GPU

Draw lines

Transform each vertex by joint model matrix *in Shader*

Creating the Joint Geometry

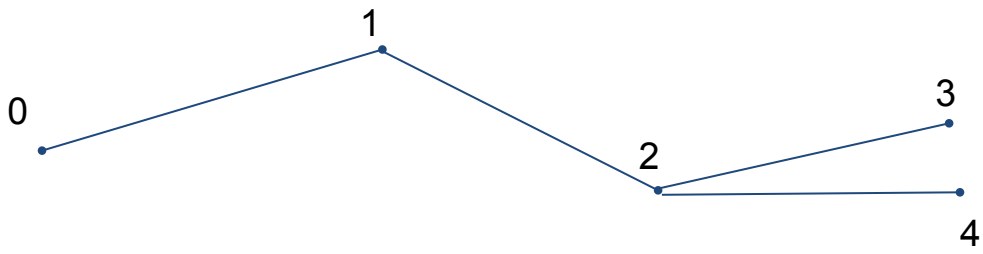
Vertex buffer is easy: *x* vertices, each with position (0,0,0)!

Index buffer requires more thought:

GL_LINES takes *pairs of lines*.

So to draw this chain our index buffer should be:

0,1, 1,2, 2,3, 2,4



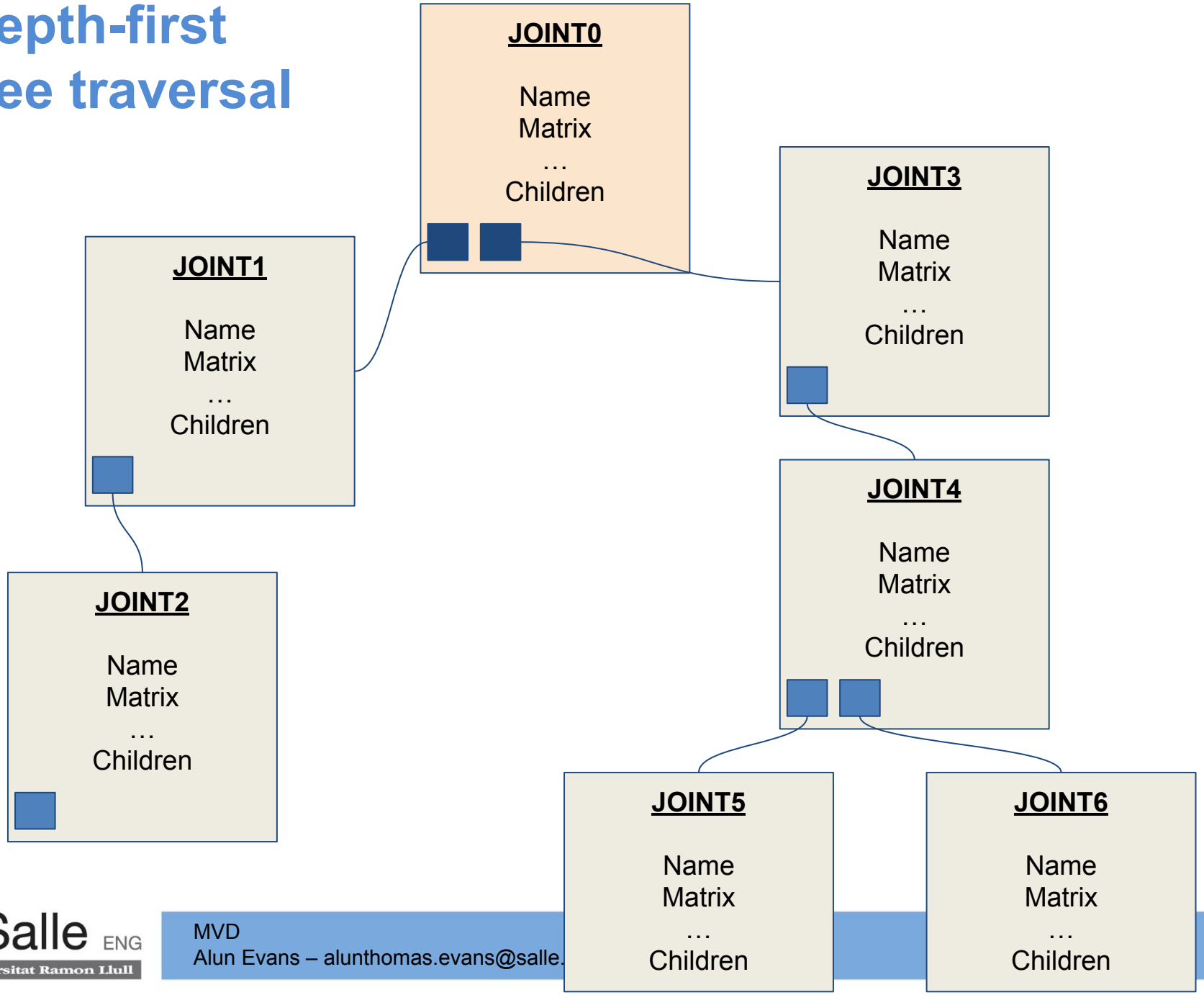
Getting the model matrices

Either way we need to get the model matrices of our joints.

But we only have a pointer to the root joint!

How do we access all of the matrices of the chain?

Depth-first tree traversal



Depth-first traversal in code

```
function traverseJointTree(Joint current) {

   //do something with current
   for (child : current->children)
      traverseJointTree(child
}
```

Time to write code!

We will draw the joints in the DebugSystem, as we don't really want to see them in final render

DebugSystem.h

```
//joints
std::vector<GLuint> joints_vaos_; // store chain VAO  might have multiple chains!
std::vector<GLuint> joints_chain_counts_; //number of joints in each chain
void createJointGeometry_(); //function to create VAOs for chains
```

TODO: createJointGeometry

For each joint chain component:

Vertex positions: create a vector<float> of size joint_chain_length * 3

Vertex indices: each joints already has a ***index_in_chain*** variable. Do **depth first traversal** (with createJointIndexBuffer) to add each index to a vector<GLuint>.

Upload to GPU!

Joint Shader

```
layout(location = 0) in vec3 a_vertex;
const int MAX_JOINTS = 96;
uniform mat4 u_model[MAX_JOINTS];
uniform mat4 u_vp;


void main(){

    gl_Position = u_vp * u_model[gl_VertexID] * vec4(a_vertex, 1.0);
}
```

gl_VertexID is an implicit openGL variable:

Name

gl_VertexID — contains the index of the current vertex

Drawing joint

I have given you a DebugSystem::drawJoint skeleton function which is called every frame:

```
//function that draws joints to screen
void DebugSystem::drawJoints_() {

    //joint shader
    glUseProgram(joint_shader_->program);

    Camera& cam = ECS.getComponentInArray<Camera>(ECS.main_camera);
    auto& jointchains = ECS.getAllComponents<JointChain>();

    //skinned_meshes size must be same as joints_vaos size
    for (size_t i = 0; i < jointchains.size(); i++) {
        // draw joint chain

    }
}
```

Drawing joints

Need to upload to shader every frame:

- View Projection of camera
- Model matrices for all joints

Can upload single float array with all matrices for all joints

```
void glUniformMatrix4fv( GLint location,
                         GLsizei count,
                         GLboolean transpose,
                         const GLfloat *value);
```

value* is a pointer to a float array of size **16 * num_joints

getJointWorldMatrices()

Depth First traversal

Receives vector<float> of size 16*num_joints

function copies content of current joint matrix into array at correct location

also need to to store **global parent model matrix**!

Drawing GL_LINES

Bind vao then call glDrawElements with GL_LINES:

```
glBindVertexArray(joints_vaos_[i]);
glDrawElements(GL_LINES, jointchains[i].num_joints * 2 , GL_UNSIGNED_INT, 0);
```

second parameter is number of elements i.e. 2 vertices per line

TODO:

Write Joint Shader

Complete getJointWorldMatrices

Complete drawJoints_()

Now plugin Animation System

Rewrite animation system to update matrix of every joint based on current frame

Need to write another depth-first recursive function to update all joints!

Currently joint chain doesn't have a ms/frame (I haven't yet exported it from Collada), so you can use a static 30fps for now

Advanced task

Plugin Transform component, so joint chain is position according to model matrix