Game Engine Architecture

Chapter 12

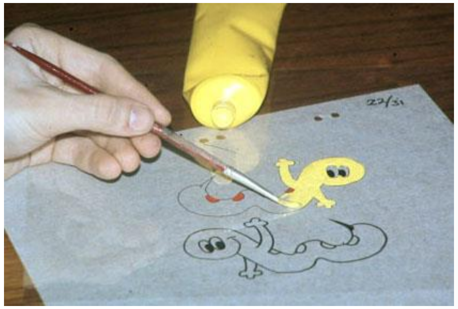
Animation Systems

Animation Systems

- The task of imbuing characters with natural-looking motion is handled by an engine component known as the *character animation system*.
- Any game object that is not 100% rigid can take advantage of the animation system.
- Example: a vehicle with moving parts, a piece of articulated machinery, trees waving gently in the breeze or even an exploding building in a game
- The three most-common techniques used in modern game engines:
  - Cel Animation, Rigid Hierarchical Animation, Per-Vertex Animation and Morph Targets

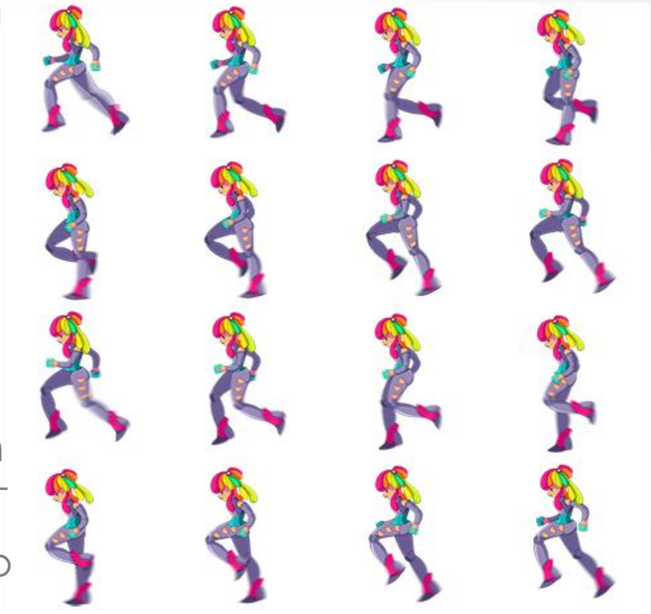
Cel animation

- *Cel animation* is a specific type of traditional animation.
- A *cel* is a transparent sheet of plastic on which images can be painted or drawn.
- An animated sequence of cels can be placed on top of a fixed background painting or drawing to produce the illusion of motion without having to redraw the static background over and over.
- The electronic equivalent to cel animation is a technology known as *sprite animation*.



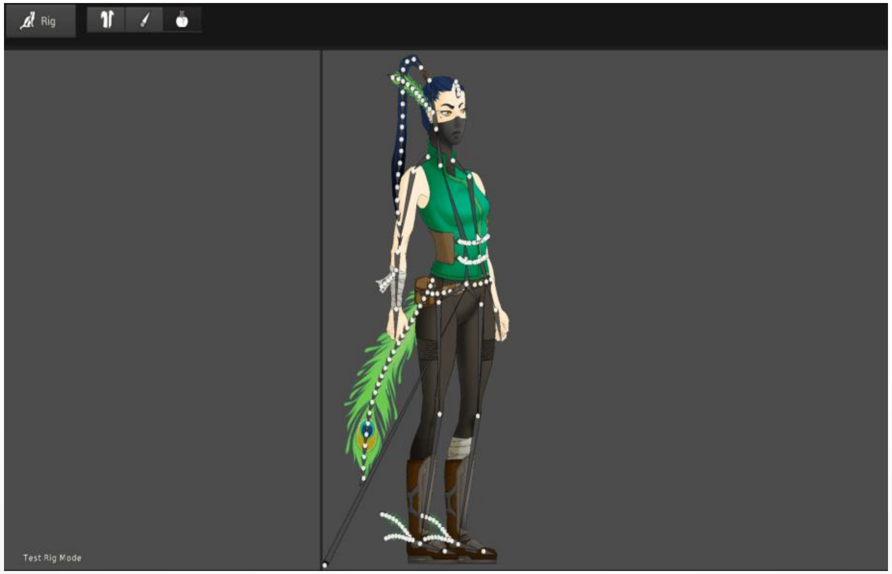
sprite animation

- A sprite is a small bitmap that can be overlaid on top of a full-screen background image without disrupting it, often drawn with the aid of specialized graphics hardware.
- A sprite is to 2D game animation what a cel was to traditional animation.
- This technique was a staple during the 2D game era.
- The sequence of frames was designed so that it animates smoothly even when it is repeated indefinitely—this is known as a *looping animation*.
- This particular animation would be called a *run cycle*
- Early 3D games like *Doom* uses a sprite-like animation system: Its monsters were nothing more than camera-facing quads, each of which displayed a sequence of texture bitmaps (known as an *animated texture*) to produce the illusion of motion.
- This technique is still used today for low-resolution and/or distant objects—for example crowds in a stadium, or hordes of soldiers fighting a distant battle in the background



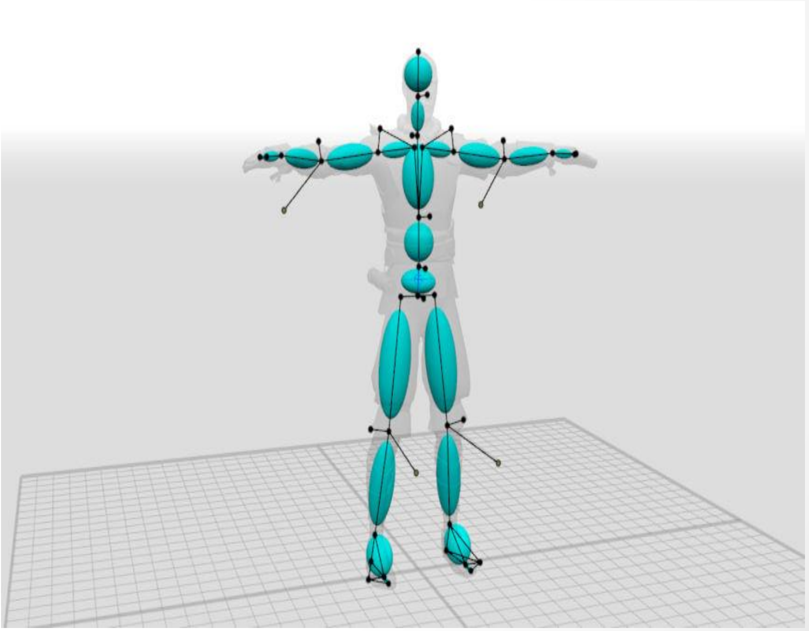
Rigid Hierarchical Animation

- A character is modeled as a collection of rigid pieces.
- The rigid pieces are constrained to one another in a hierarchical fashion, analogous to the manner in which a mammal's bones are connected at the joints.
- For example, when the upper arm is moved, the lower arm and hand will automatically follow it.
- The big problem with the rigid hierarchy technique is that the behavior of the character's body is often not very pleasing due to "cracking" at the joints.
- Rigid hierarchical animation works well for robots and machinery that really are constructed of rigid parts, but it breaks down under scrutiny when applied to "fleshy" characters.



Rigid Hierarchical Animation

- Pelvis
    - Torso
        - UpperRightArm
            - LowerRightArm
                - RightHand
        - UpperLeftArm
            - LowerLeftArm
                - LeftHand
        - Head
    - UpperRightLeg
        - LowerRightLeg
            - RightFoot
    - UpperLeftLeg
        - LowerLeftLeg
            - LeftFoot



Per-Vertex Animation

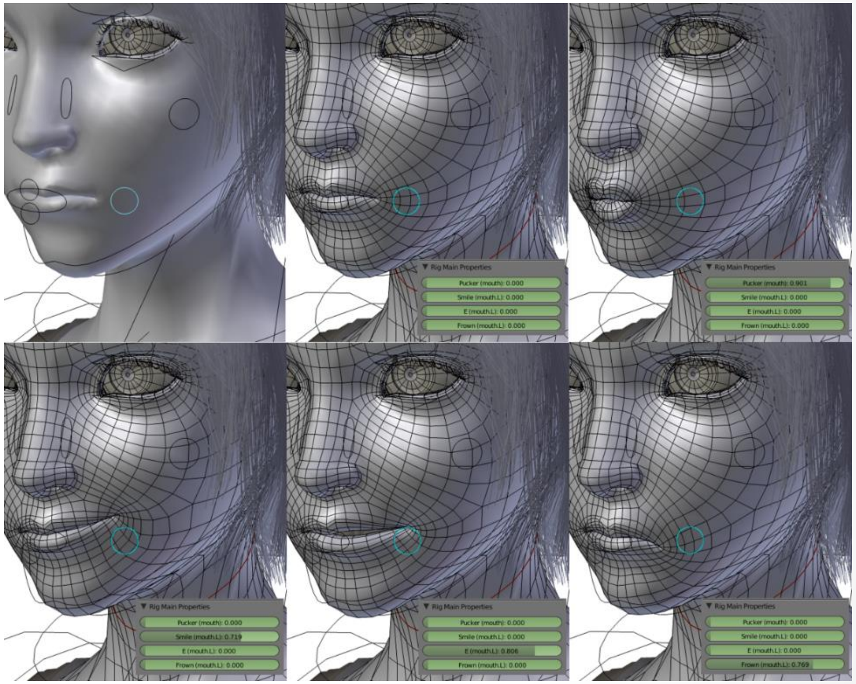
- What we really want is a way to move individual vertices so that triangles can stretch to produce more natural-looking motion.
- *Per-vertex animation:*the vertices of the mesh are animated by an artist, and motion data is exported, which tells the game engine how to move each vertex at runtime.
- It is a data-intensive technique, since time-varying motion information must be stored for each vertex of the mesh.
- it has little application to real-time games.

Morph target animation

- The vertices of a mesh are moved by an animator to create a relatively small set of fixed, extreme poses.
- Animations are produced by *blending* between two or more of these fixed poses at runtime.
- The position of each vertex is calculated using a simple linear interpolation (LERP) between the vertex's positions in each of the extreme poses.
- Here is the **vertex shader** in action: a morphing between a torus and a cylinder.
- **Morph target animation** is a technique that allows to deform a mesh using different deformed versions of the original mesh.
- This technique is used in character animation for example. The deformed versions are the **morph targets** (also called **blend shapes**).
- The deformation from one morph target to another one is done by interpolating the vertex positions.
- Here is a simple morph target animation technique that stores the vertex positions of the morph targets in the original mesh using vertex attribute arrays
- All morph targets must have the same number of vertices. All animation work is achieved in the vertex shader.

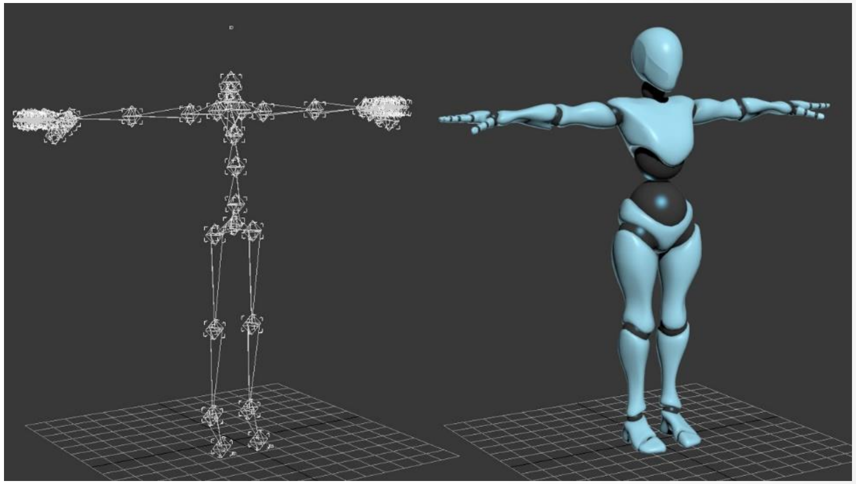
Facial animation

- The morph target technique is often used for facial animation, because the human face is an extremely complex piece of anatomy, driven by roughly 50 muscles.
- Morph target animation gives an animator full control over every vertex of a facial mesh to produce both subtle and extreme movements that approximate the musculature of the face well.
- As computing power continues to increase, some studios are using jointed facial rigs containing hundreds of joints as an alternative to morph targets.
- Other studios combine the two techniques, using jointed rigs to achieve the primary pose of the face and then applying small tweaks via morph targets.



Skinned Animation

- Skinned animation was first used by games like *Super Mario 64*, and it is still the most prevalent technique in use today
- a *skeleton* is constructed from rigid "bones," just as in rigid hierarchical animation
- However, instead of rendering the rigid pieces on-screen, they remain hidden.
- A smooth continuous triangle mesh called a *skin* is bound to the joints of the skeleton.
- Each vertex of the skin mesh can be weighted to multiple joints, so the skin can stretch in a natural way as the joints move

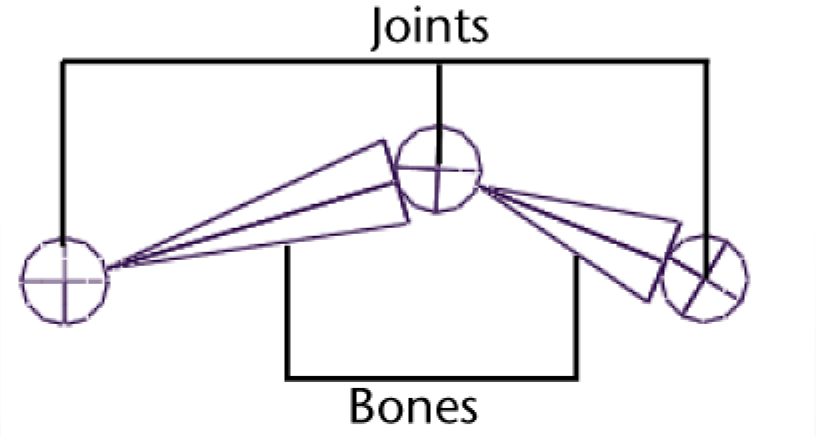


Animation Methods as Data Compression Techniques

- *Compressing* the amount of information needed to describe an animation by restricting ourselves to moving only the vertices.
- Morph targets can be thought of as an additional level of compression, achieved by imposing additional constraints on the system
- Vertices are constrained to move only along linear paths between a fixed number of predefined vertex positions.
- Skeletal animation is just another way to compress vertex animation data by imposing constraints.

Skeletons

- A skeleton is comprised of a *hierarchy* of rigid pieces known as *joints*.
- the joints are the objects that are directly manipulated by the animator,
- while the bones are simply the empty spaces between the joints.
- Game engines don’t care about bones—only the joints matter.
- Whenever you hear the term “bone” being used in
- the industry, remember that 99% of the time we are actually speaking about joints.



The Skeletal Hierarchy

- The joints in a skeleton form a hierarchy or tree structure.
- We usually assign each joint an index from 0 to N - 1. Because each joint has one and only one parent, the hierarchical structure of a skeleton can be fully described by storing the index of its parent with each joint.
- The root joint has no parent.

Representing a Skeleton in Memory

- Each joint data structure typically contains the following information:
- The *name* of the joint, either as a string or a hashed 32-bit string id.
- The *index* of the joint's *parent* within the skeleton.
- The *inverse bind pose transform* of the joint. The bind pose of a joint is the position, orientation and scale of that joint at the time it was bound to the vertices of the skin mesh.

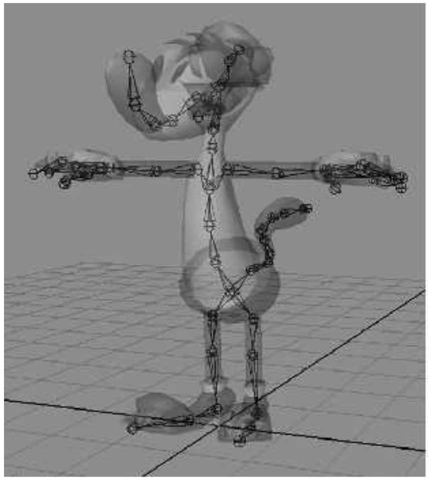
```
struct Joint
{
Matrix4x3 m_invBindPose; // inverse bind pose  transform
const char* m_name; // human-readable joint name
U8 m_iParent; // parent index or 0xFF if root
};
struct Skeleton
{
U32 m_jointCount; // number of joints
Joint* m_aJoint; // array of joints
};
```

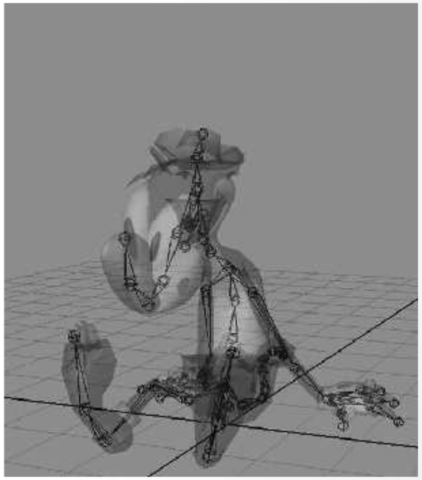
Poses

- In skeletal animation, the pose of the skeleton directly controls the vertices of the mesh, and posing is the animator's primary tool for breathing life into her characters.
- Before we can animate a skeleton, we must first understand how to *pose* it.
- A skeleton is posed by rotating, translating and possibly scaling its joints in arbitrary ways.
- A joint pose is usually represented by a 4x4 or 4x3 matrix, or by an SRT data structure (scale, quaternion
- rotation and vector translation).
- The pose of a skeleton is just the set of all of its joints' poses and is normally represented as a simple array of matrices or SRTs.

Bind Pose

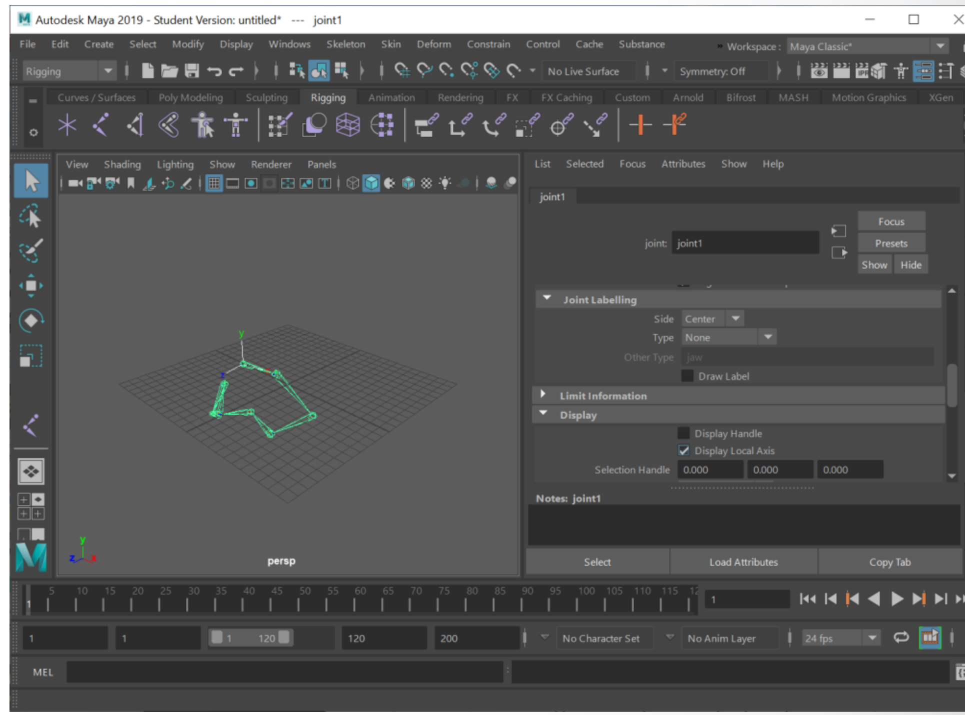
- The pose on the left is a special pose known as the *bind pose*, also sometimes called the *reference pose* or the *rest pose*.
- This is the pose of the 3D mesh prior to being bound to the skeleton (hence the name).
- Bind pose is the pose that the mesh would assume if it were rendered as a regular, unskinned triangle mesh, without any skeleton at all.
- The bind pose is also called the *T-pose* because of the shape of letter T.





Local Poses

- We use the term *local pose* to describe a parent relative pose.
- Local poses are almost always stored in SRT format.
- Maya represent joints as small spheres.
- Maya gives the user the option of displaying a joint's local coordinate axes



Joint Scale

- using a lower-dimensional scale representation can save memory.
- Uniform scale requires a single floating-point scalar per joint per animation frame, while nonuniform scale requires three floats, and a full 3x3 scale-shear matrix requires nine.
- Restricting our engine to uniform scale has the added benefit of ensuring that the bounding sphere of a joint will never be transformed into an ellipsoid, as it could be when scaled in a nonuniform manner.

Representing a Joint Pose in Memory

```
struct JointPose
{
Quaternion m_rot; // R
Vector3 m_trans; // T
F32 m_scale; // S (uniform scale only)
};
//If nonuniform scale is permitted, we might define a
joint pose like this instead:
struct JointPose
{
Quaternion m_rot; // R
Vector4 m_trans; // T
Vector4 m_scale; // S
};
```

Local Pose

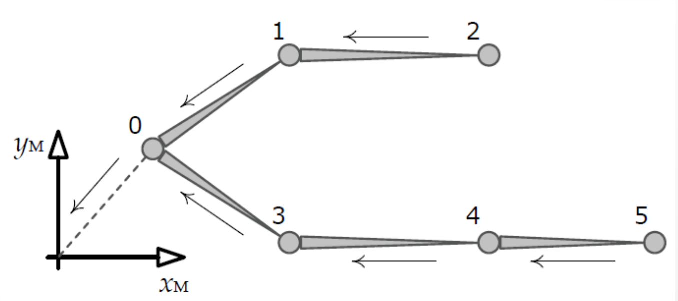
- The local pose of an entire skeleton can be represented as follows:
- array m_aLocalPose is dynamically allocated to contain just enough occurrences of JointPose to match the number of joints in the skeleton.

```
struct SkeletonPose
{
Skeleton* m_pSkeleton; // skeleton + num joints
JointPose* m_aLocalPose; // local joint poses
};
```

Global Poses

- Sometimes it is convenient to express a joint's pose in model space or world space. This is called a *global pose*.
- A global pose can be calculated by walking the hierarchy from the joint in question towards the root and model-space origin, concatenating the child-to-parent (local) transforms of each joint as we go.

```
struct SkeletonPose
{
Skeleton* m_pSkeleton; // skeleton + num joints
JointPose* m_aLocalPose; // local joint poses
Matrix44* m_aGlobalPose; // global joint poses
};
```



Clips

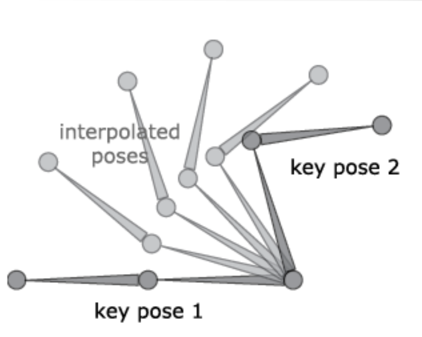
- A game character's movement must be broken down into a large number of fine-grained motions.
- We call these individual motions *animation clips*, or sometimes just *animations*.
- Each clip causes the character to perform a single well-defined action.
- Some clips are designed to be looped—for example, a walk cycle or run cycle.
- Others are designed to be played once—for example, throwing an object or tripping and falling to the ground.
- Some clips affect the entire body of the character—the character jumping into the air for instance.

Noninteractive and semi-interactive sequences

- Sometimes game characters are involved in a noninteractive portion of the game, known as an *in-game cinematic* (IGC), *noninteractive sequence* (NIS) or *full-motion video* (FMV).
    - The terms IGC and NIS typically refer to noninteractive sequences that are rendered in real time by the game engine itself.
    - The term FMV applies to sequences that have been prerendered to an MP4, WMV or other type of movie file and are played back at runtime by the engine's full-screen movie player.
- semi-interactive sequence
    - Known as a *quick time event* (QTE). In a QTE, the player must hit a button at the right moment during an otherwise noninteractive sequence in order to see the success animation and proceed;
    - otherwise, a failure animation is played, and the player must try again.

Pose Interpolation and Continuous Time

- the rate at which frames are displayed to the viewer is not necessarily the same as the rate at which poses are created by the animator.
- In both film and game animation, the animator almost "never" poses the character every 1/30 or 1/60 of a second.
- Instead, the animator generates important poses known as *key poses* or *key frames* at specific times within the clip, and the computer calculates the poses in between via linear or curve based interpolation.
- Figure shows how an animator creates a relatively small number of key poses, and the engine fills in the rest of the poses via interpolation.
- Because of the animation engine's ability to *interpolate* poses, we can actually sample the pose of the character at *any time* during the clip.
- Animations are sometimes *time-scaled* in order to make the character appear to move faster or slower than originally animated.
- In a real-time game, an animation clip is almost *never* sampled on integer frame numbers.



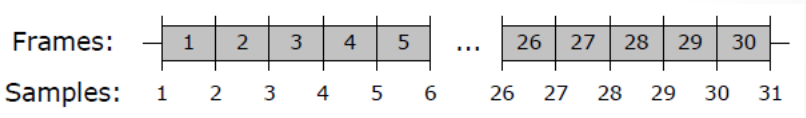
Time Units

- measured in units of seconds.
- Time can also be measured in units of *frames*.
- Typical frame durations are 1/30 or 1/60 of a second for game animation.
- t should be a real (floating point) quantity, a fixed-point number or an integer that measures very small subframe time intervals.

Frame versus Sample

- *sample* to refer to a single point in time,
- *frame* to describe a time period that is 1/30 or 1/60 of a second in duration.

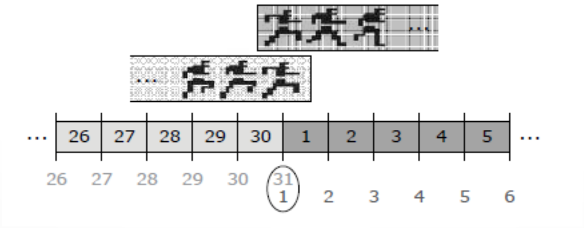
- So for example, a one-second animation created at a rate of 30 frames per second would consist of 31 *samples* and would be 30 *frames* in duration.



- The term "sample" comes from the field of signal processing. A continuous-time signal (i.e., a function *f* (*t*)) can be converted into a set of discrete data points by sampling that signal at uniformly spaced time intervals.

Frames, Samples and Looping Clips

- When a clip is designed to be played over and over repeatedly, we say it is *looped*.
- Imagine two copies of a 1 s (30-frame/31-sample) clip laid back-to-front, then sample 31 of the first clip will coincide exactly in time with sample 1 of the second clip.
- For a clip to loop properly, then, we can see that the pose of the character at the end of the clip must exactly match the pose at the beginning.

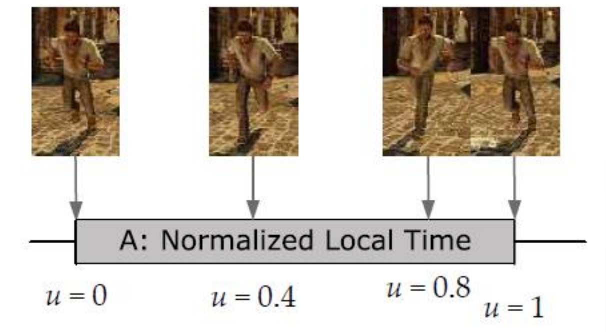


The last sample

- the last sample of a looping clip (in our example, sample 31) is redundant.
- Many game engines therefore remove the last sample of a looping clip.
  - If a clip is *non-looping*, an *N*-frame animation will have $N + 1$ unique samples.
  - If a clip is *looping*, then the last sample is redundant, so an *N*-frame animation will have *N* unique samples.

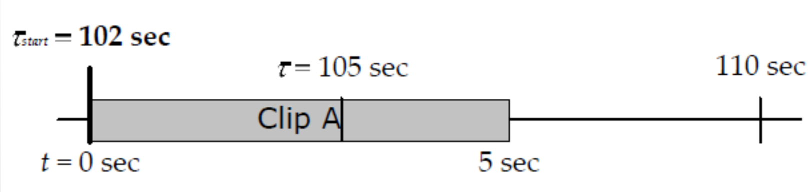
Normalized Time (Phase)

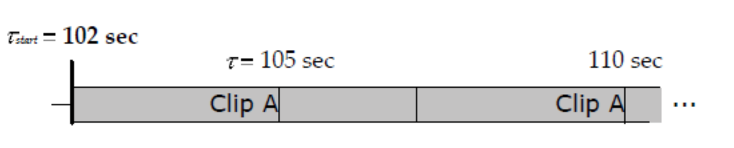
- a normalized time unit u, such that $u = 0$ at the start of the animation, and $u = 1$ at the end, no matter what its duration T may be.
- u acts like the phase of a sine wave when the animation is looped.
- Useful when synchronizing two or more animation clips that are not necessarily of the same absolute duration.
- For example, we might want to smoothly cross-fade from a 2-second (60-frame) run cycle into a 3-second (90-frame) walk cycle.
- We can accomplish this by simply setting the normalized start time of the walk clip, u_{walk}, to match the normalized time index of the run clip, u_{run}.
- We then advance both clips at the same normalized rate so that they remain in sync



The Global Timeline

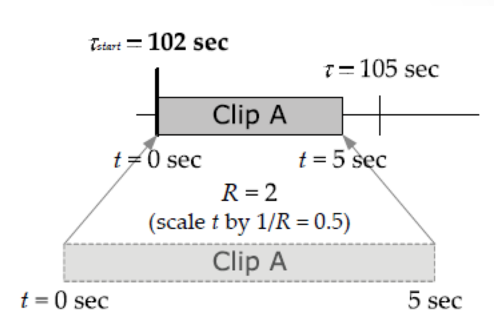
- every animation clip has a local timeline (whose clock starts at 0 at the beginning of the clip)
- every character in a game has a global timeline (whose clock starts when the character is first spawned into the game world)
- *Playing* an animation is simply *mapping* that clip's local timeline onto the character's global timeline.
- Playing a looping animation is like laying down an infinite number of back-to-front copies of the clip onto the global timeline

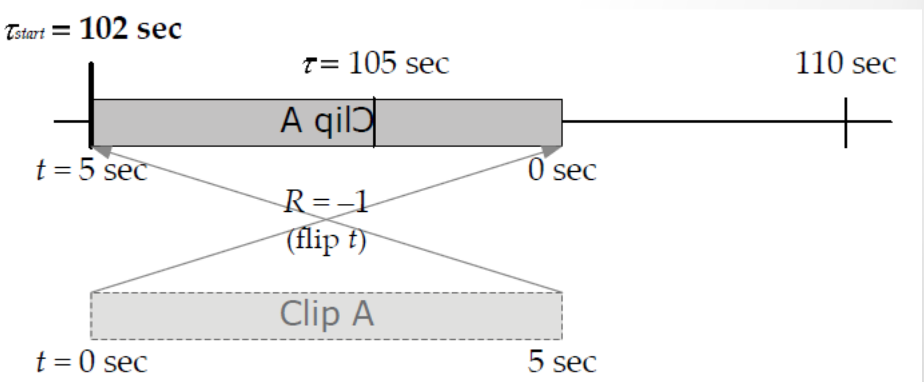




Time-scaling

- *Time-scaling* a clip makes it appear to play back more quickly or more slowly than originally animated.
- Time-scaling is most naturally expressed as a *playback rate*.
- For example, if an animation is to play back at twice the speed (R = 2), then we would scale the clip's local timeline to one-half (1/R = 0.5) of its normal length when mapping it onto the global timeline.
- Playing a clip in reverse corresponds to a time scale of -1.





Comparison of Local and Global Clocks

- The animation system must keep track of the time indices of every animation that is currently playing. Two choices:
  - each clip has its own local clock,
  - the character has a global clock
- The local clock approach has the benefit of being simple, and it is the most obvious choice when designing an animation system.
- the global clock approach is good when it comes to synchronizing animations, either within the context of a single character or across multiple characters in a scene.

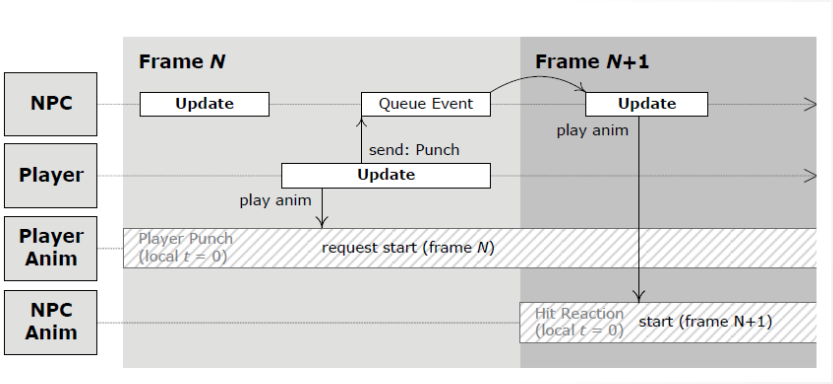
Synchronizing Animations with a Local Clock

- With a local clock approach, the origin of a clip's local timeline ($t = 0$) is usually defined to coincide with the moment at which the clip starts playing.
- To synchronize two or more clips, they must be played at exactly the same moment in game time
- tricky when the commands used to play the animations are coming from different engine subsystems.
- For example, let's say we want to synchronize the player character's punch animation with a non-player character's corresponding hit reaction animation.
- The problem is that the player's punch is initiated by the player subsystem in response to detecting that a button was hit on the joy pad.
- Meanwhile, the non-player character's (NPC) hit reaction animation is played by the artificial intelligence (AI) subsystem.
- If a message-passing (event) system is used to communicate between the two subsystems, additional delays might be incurred

Synchronizing Animations

- The order of execution of different gameplay systems can introduce animation synchronization problems when local clocks are used.

```
void GameLoop()
{
while (!quit)
{
// preliminary updates
UpdateAllNpcs(); // react to punch
event  from last frame
// more updates...
UpdatePlayer(); // punch button hit
- start punch
// anim, and send event to NPC to
// react
// still more updates...
}
}
```



Synchronizing Animations with a Global Clock

- A global clock approach helps to alleviate many of these synchronization problems, because the origin of the timeline ($t = 0$) is common across all clips
- We need to ensure that the two characters' global clocks match!
- We can either adjust the global start times to take account of any differences in the characters' clocks, or we can simply have all characters in the game share a single master clock.