Lecture 17: Graphical Simulation of Fire

Amorphous objects, such as fire, smoke, water, mist and fog are difficult subjects for graphical rendering since they do not have well defined boundaries, and are naturally mobile. They are currently the subject of considerable research and development because of their commercial importance in the film and games industries. Recent systems, using fluid mechanics have created some very impressive effects, but at high computational cost. These are being deployed in film, where each frame of an animation may be computed off-line. A major challenge remains in real time computer graphics, where effects must be computed at frame rate (typically 25 times/second). In this lecture we will look at fire as an exemplar, and discuss some of the approaches that have been taken to simulate it visually.

The simplest way to create a fire is to hand draw a set of polygons which can be overlaid on an image. This is a time consuming and unadaptable procedure, but it can be quite effective for special effects, such as short lived explosions or gun shot. If a sequence of polygons is used in isolation the effect is unconvincing because fire emits light and will illuminate the surrounding polygons of the scene. A further refinement is to place one or more light sources around the position of the fire, and to control their illumination settings to synchronise with the polygons representing the fire. This adds realism at little computational cost, and so is suitable for real time use in computer games. There are obvious limitations of polygon fires. They are difficult to sustain for any length of time without intrusive repeats. Changes of shape, and spreading of the fire are difficult to create, and translucency of the flames cannot be easily created.

Particle Models

A better approach to creating fire effects is to try to model the underlying physical process more accurately. Flames are incandescent gases, formed where a combustible gas has an interface with air, whose visual appearance depends on their temperature, pressure and density. The dynamics of flames is determined by diffusion. As the gases diffuse they become less dense and they cool introducing changes of colour and translucency. Clearly this is a complex process, and exact modeling is not possible. However, approximate models can be constructed to create good visual effects.

One way in which physicists model gases is as a collection of particles, and this is indeed the approach that has been taken for most computer graphics systems. Particles form a discrete approximation to a continuous material, and have the advantage that they can be highly mobile, creating shape changes. They can given stochastic behaviour properties to create animations that do not repeat regularly.

As a simple illustration (using water rather than fire), consider the case of a jet of water. At low resolution we can think of the particles as being equivalent to droplets. Each has a mass and is ejected from the nozzle with a particular velocity. Once moving freely in the air each droplet is acted on by gravity and perhaps wind forces and its behaviour is governed by Newtonian mechanics. Simple equations determine the path of each particle during one frame interval, and they can be rendered by simple streaks, or blobs as shown in Figure 1. Using this technique pleasing visual effects can be created with very little computation.

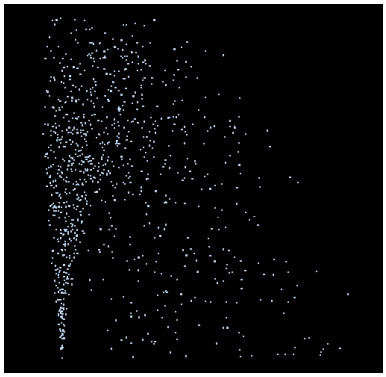


Figure 1: Particle Fountain

For modeling fire, particles can have a number of attributes which include:

Position	[x,y,z]
Velocity	[u,v,w]
Mass	m
Temperature	θ
Lifetime	t

Most systems represent particles as points or small spheres, but it might also be possible to include shape as a attribute. Particles can be created with some stochastic elements to provide variability, and can be made to move using physical rules. They can be given a lifetime, and the can be rendered using a number of techniques to create the effects of flame or smoke. The simplest way to move particles is to give them Newtonian dynamics, that is to say they conform to Newtons first law: $\mathbf{f} = m\mathbf{a}$ where $\mathbf{f}$ is a vector force, and $\mathbf{a}$ is a vector acceleration.

To solve for the particle's motion we need to integrate using:

$$\mathbf{a} = \frac{d\mathbf{v}}{dt} = \frac{d^2\mathbf{x}}{dt^2}$$

where $\mathbf{v}$ is the velocity and $\mathbf{x}$ is the position. Since we are working on a fixed frame interval we can carry out the integration using a simple numerical approximation:

$$\mathbf{v_{t+1}} = \mathbf{a_t}\Delta t + \mathbf{v_t} \qquad \mathbf{x_{t+1}} = \mathbf{v_t}\Delta t + \mathbf{x_t}$$

An easy extension of this is to introduce damping (or friction) into the equations. This is appropriate for gases since it describes the viscous drag on each particle:

$$\mathbf{f} = m\mathbf{a} + s\mathbf{v}$$

Another possible extension is to allow particles to collide. This however is much harder to do and is computationally very intensive. It requires sub frame calculation, and some specification of the particle size and shape. It involves a large number of particle/particle checks. The effect of collisions can be simulated by applying small random forces to a proportion of the particles at each frame interval.

Particle rendering is a major component in creating a realistic fire. Some possible strategies are (i) to make the colour age dependent (ii) make the colour temperature dependent. The latter strategy means that it is necessary to have some physical modeling that accounts for particle cooling.

Exemplar: The Wrath of Khan

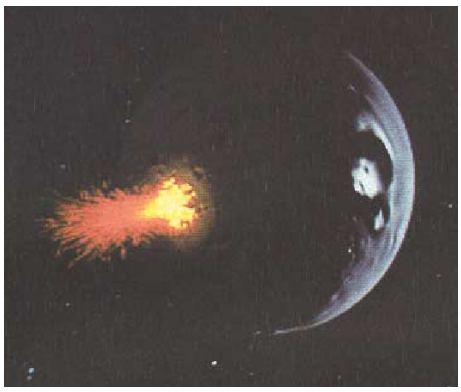


Figure 2: The Explosion in the Wrath of Khan (Reeves(1983))

The first major use of a particle system for creating fire was in the star trek film "The Wrath of Khan" and was published by Reeves et al (SIGGRAPH 1983 17(3) 359-376). The sequence depicts a planet being brought to life by exploding a bomb on it. The particles were emitted from locations arranged in concentric rings around the point of impact of the bomb. By using a timed activation sequence for the particle systems the effect of flames spreading could be simulated. In total 750,000 particles were used - this was a huge number in 1982, but is quite modest in todays terms. The particles were generated with velocity directions distributed normally about the normal vector to the surface as shown in figure 3. The velocity magnitude was also normally distributed. Particles were slowed down by a gravitational force, but extinguished after a period of time so that their parabolic path was not apparent. Each particle was rendered by a short line segment whose colour was selected to simulate the colour characteristics of different temperatures. Yellow represents the hottest, cooling to red through the particle life time. Since the particles did not emit light themselves, a fake light source was placed above the point of impact on the planet. This can be seen clearly in figure 2.

The fire in The Wrath of Khan was certainly groundbreaking at the time, but it fell short of creating a realistic visual simulation in two respects. First rendering using simple line streaks is unnatural. The colouring successfully fools the eye into accepting that what can be seen is a flame, but the flames do not travel in straight lines. In practice the particles can describe the presence of a combustible gas, but the flame occurs at the boundary of the gas, where it combines with oxygen to burn. Thus rendering each particle individually will not produce a good result. Secondly the motion of the particles is not realistic. Burning gases are lighter than air and so tend to rise and spread out rather than fall under gravitational forces.

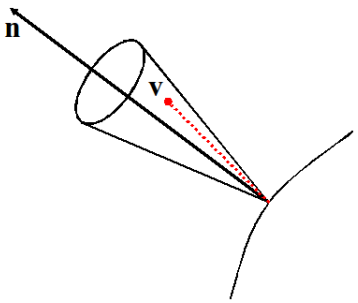


Figure 3: Generating Particles

Improved rendering: Bitmap Splatting

Bitmap splatting is a technique that involves projecting particles onto the raster, and then rendering them by blending a small and appropriate bitmap with the pixels already rendered. For flames the bitmaps are made up of appropriately coloured pixels, with a higher proportion of yellow for hotter regions, and a high proportion of black (invisible) as the particles cool. The blending allows the possibility of flames being translucent, and makes the visual stimuli correspond better to the visual characteristic of fire. Bitmap splats are also very fast to render which makes them attractive for fires created with small numbers of particles.

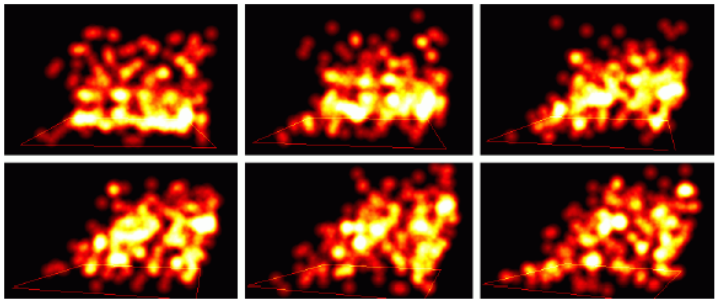


Figure 4: Bitmap Splat fire by Engell-Nielsen and Madsen, University of Copenhagen

Improved Rendering: Ray Tracing Particle Systems

An improvement in rendering can be achieved using ray casting or ray tracing. As before the flame dynamics is modeled by the particles, but the rendering now takes into account the number and density of the particles without actually rendering them directly. For ray casting, a blob (usually a sphere) is placed around each particle. A ray is cast from each pixel and out into the particle system. If it passes through a blob, it takes on a a proportion of the blob colour which depends on the temperature of the blob. A ray that misses all the blobs will be rendered with the colour of the background object it hits. A ray passing through a few blobs will be rendered with a blend of the colours collected from the blobs and the background, and a ray passing through a dense region of particles will take its entire colour from the particles. This will create effects such as translucency of the flames.

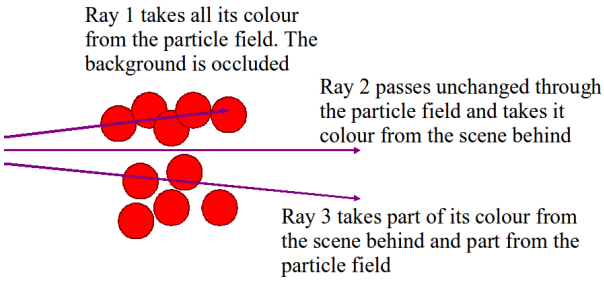


Figure 5: Ray Casting through flames

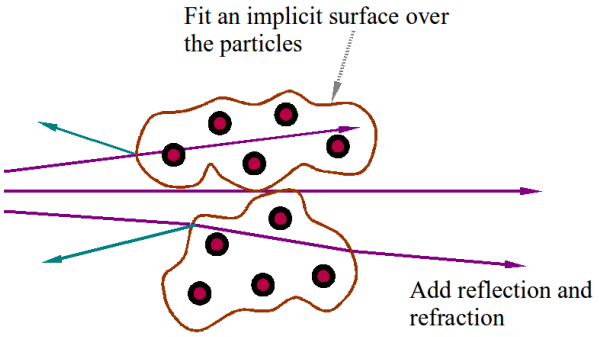


Figure 6: Ray Tracing Flames

A better approach is to use implicit surfaces rather than blobs. A simple way to fit an implicit surface is to consider each particle a point charge, and then calculate an iso-potential surface. This provides a better model since the implicit surface can be considered the interface between the burning gas and the air. The particle motion still controls the dynamics of the flame, but the rendering represents the interface between the combustible gas and the air more realistically. Ray colours can be accumulated as before, with a temperature being computed for each point on the implicit surfaces, and there is now the possibility of calculating refractions and reflections of the rays for even better effects, though this will be a time consuming process.

A further step in adding realism is to use radiosity to make the flame illuminate the surrounding objects. This is a major computation task, since the particles move in each frame and therefore the form factors need

to be re-computed at each step. In order to do this we need to create the implicit surface and then tessellate it using polygons, for example triangles.

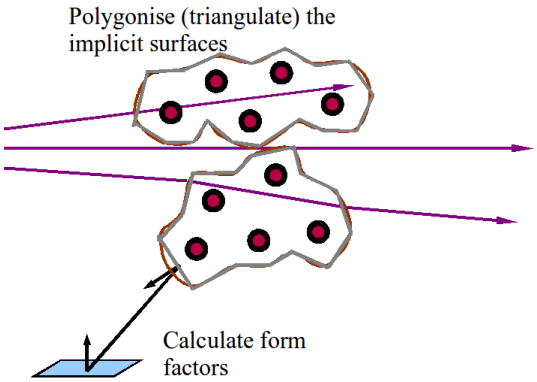


Figure 7: Radiosity effects in rendering flames

Improved motion

One of the big challenges in simulating fire is making the particles move in a plausible manner. There are a number of heuristic techniques for doing this, and most are based on the creation of a force (or velocity) field in the volume in which the particle system exists. Force fields can create effects such as diffusion. Gases diffuse from more dense regions to less dense ones and cool as part of the process. This process is computed in a discrete manner by dividing the volume where the fire is burning into voxels. At any one time the pressure in a voxel can be taken as proportional to the number of particles in it, and diffusion forces can be applied to each particle before integrating to calculate the velocity and the motion. A further refinement is to remove particles as they come close to the boundary (implicit surface), which is where the gas burns to form CO_2.

If a velocity field is used then particles starting from any single point will follow a streamline, which can be found by integration. A point (or particle) in a velocity field will change it's position in frame time by $\mathbf{v}\Delta(t)$ where $\mathbf{v}$ is the velocity at its position in the field. Particles, depending on their positions follow different streamlines, and the combined motion can be used to create the effect of the motion of the incandescent gas. Velocity fields on their own produce static streamlines, and therefore could be too repetitive to create a pleasing visual effect. They could be used in conjunction with force fields to create more varied motion.

More recently, there has been an interest in using fluid dynamics, described by the Navier-Stokes equation, to determine how the gas actually flows, and to create effects such as swirling and vortices which can often be seen in flames. The Navier-Stokes equation is a differential equation connecting the force, velocity, viscosity and pressure at each point within a fluid. It is usual to assume that the fluid is incompressible, and even with flames this assumption gives reasonable results. Numerical solutions to the Navier-Stokes equation can be found for fixed boundary conditions. They could be used to create a dynamic velocity field in 3D which can be used to update the particles, or to solve directly for the gas air interface. Methods based on the Navier-Stokes equation are beyond the scope of this course. More details can be found in: Nguyen, Enright and Fedkiw - Simulation and Animation of Fire.

Lecture 18: Kinematics and Animation

Animation of 3D Models

Figure 1: King Kong (from Visions in B & W)

In the early days of cinema 3D animation was achieved using a physical 3D model which was adjusted by hand to create each individual frame of the animation. The classic example of the use of this technique was in the movie King Kong (1933), in which the model of Kong was only about one foot high. Animation created by this technique is still popular, a recent example being Wallace and Gromit.

Computer support systems for animation began to appear in the late 1970s, and the first computer generated 3D animated full length film was Toy Story (1995). Although completely created using the computer, Toy Story, and other modern 3D animation films are still enormously costly in human time. Fully automatic animation is still a long way off, but computer tools to support tools have developed rapidly. These remove much of the tedious work of the animator, and allow the creation of spectacular special effects. Basic approaches are: (i) Physical Models; (ii) Procedural Methods and (iii) Keyframing.

Physical models

This approach is suitable for inanimate objects or effects where there is a simple tractable physical model. Two examples are: bouncing balls and cars driving on rough roads. In both these cases the movement can be calculated simply using Newtonian mechanics. In other cases composite models can be applied, such as spring mass damper arrays for animating fluttering flags or particle systems for creating fire smoke or water.

Procedural models

The behaviour of an object is not modelled at all, but described as a procedure. The description often takes the form of a script. This kind of approach is appropriate for easily defined behaviours in inanimate objects. One example is crack propagation in glass or concrete, which can be used in a variety of effects. Crack propagation is difficult to model physically, since it is influenced by non-uniform material properties, but easy to describe procedurally since cracks tend to follow common patterns.

Keyframes

Keyframing was a traditional method in hand drawn animation where the senior animator drew the key frames at time intervals of around one second, and some stooge would then laboriously draw all the in-between frames. Systems have evolved in which the computer calculates a number of in between positions to effect a smooth movement between the two keyframes. These can be specified using, for example, spline curves to define a path of motion for some particular points on the moving object. However, even this process is difficult to automate while maintaining a plausible movement. In particular it is necessary to (broadly) observe physical laws.

One method to constrain an animation of human or animal figures is to use a jointed skeletons. The skeleton is made up of a set of links. Each link in the chain is rigid, and the movement is constrained by the degree of freedom at each joint. The skeleton can be fleshed out in any way to create the animated figure. One early example was Luxo Jr. (1987) created by Pixar (http://www.pixar.com/shorts/ljr/theater/short_320.html). This was hailed as the first computer generated 3D animation to have the same natural appearance as the older hand crafted animation. However, it still needed considerable human input to create appealing plausible movements. The reason for this is that movements of humans and animals follow rather complex patterns which are difficult to describe.

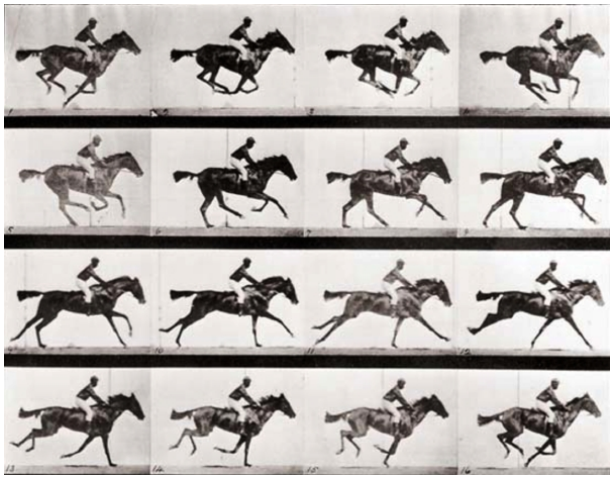


Figure 2: Galloping Horse by Eadweard Muybridge (1878)

The complexity of movement was first obesrved by Eadweard Muybridge using a set of synchronised cameras to capture frames separated by very short time intervals. His work was intended to create animted photographs, and was a forerunner of the modern cinema. His most famous set of photographs shows the various stages of a horse galloping (showing for the first time that all four legs do leave the ground at one point) and illustrates just how complex the movement actually is.

The complexity of natural movement makes it difficult to specify for computer based animation systems. For this reason it is cheaper to used motion capture in film special effects. Two recent examples of this were Gollum, in "The Lord of the Rings" trilogy, and King Kong in the 2005 (not quite as good) remake of the classic 1933 film. In both cases an actor is filmed making the required movements. Then key points are tracked throughout the sequence, and used to control the computer model's movements. In the case of King Kong the actor spent time at a zoo studying the movements made by gorillas, so that he could then imitate them. This was a cheaper process than trying to specify them in a computer readable form.

Kinematics

Although motion capture is an effective way of collecting complex information about movement, it is still very human intensive and therefore costly. Thus there is still considerable interest in studying the movement of jointed chains. The work also applies to the mechanics of robot movement and is called kinematics. One of the most interesting current problems is the study of inverse Kinematics. Here we specify the movement path of say one point on a chain (for example the foot in figure 3) and then try to reconstruct the movement of all other points that create that movement given certain constraints or possibly the need to minimise the motion as a whole. Normally the links of the articulated chain are taken as rigid, and the joints are constrained by the degrees of freedom they support. Three examples of joints found in the body are illustrated in Figure 4.

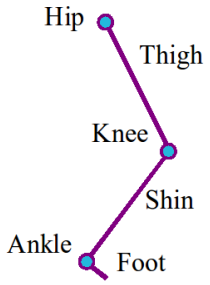
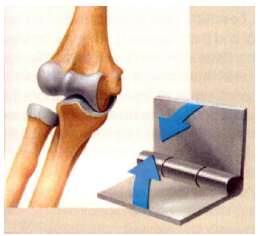
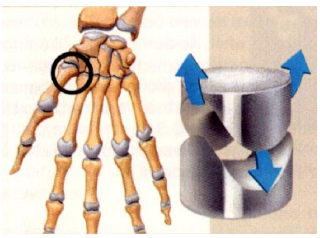


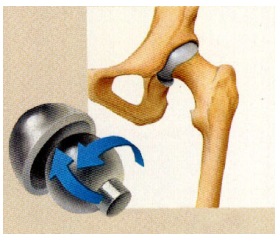
Figure 3: Articulated Chain for a Leg



(a) Hinge: 1 degree of freedom



(b) Saddle: 2 degrees of freedom



(c) Socket: 3 degrees of freedom

Figure 4: Different types of Joints (images from www.shockfamily.net)

Euler Angles

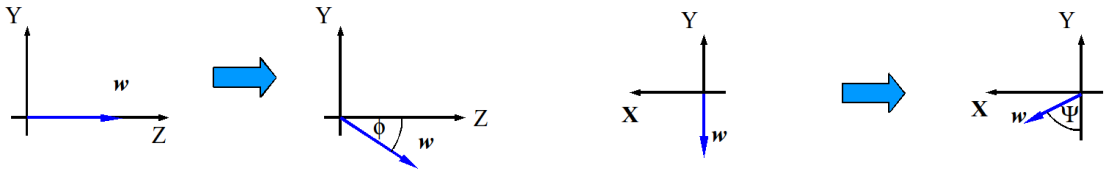
At any joint we can specify the orientation of one link relative to the other by the Euler angles. These encode pitch, roll and yaw. The links are fixed in length (rigid), so the position of end of the chain is completely defined by a set of Euler angles. For the leg shown in Figure 3 we have 3(hip) + 1(knee) + 1 (ankle) = 5 variables (Euler Angles) to determine the end point.

Euler was the first mathematician to prove that any rotation of 3D space could be achieved by three independent rotations. The rotations could be performed in many ways, but the usual convention is to: (i) Rotate about Z; (ii) Rotate about X and (iii) Rotate about Z (again) The complete Euler rotation matrix is therefore:

$$\begin{bmatrix} Cos(\theta) & Sin(\theta) & 0 & 0 \\ -Sin(\theta) & Cos(\theta) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & Cos(\phi) & Sin(\phi) & 0 \\ 0 & -Sin(\phi) & Cos(\phi) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} Cos(\psi) & Sin(\psi) & 0 & 0 \\ -Sin(\psi) & Cos(\psi) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

This formulation is not very intuitive, so to see what is happening consider transforming a vector $\boldsymbol{w}$ lying along the z axis.

The first rotation matrix will not change the vector $\boldsymbol{w}$ at all. In this case it turns the $\boldsymbol{u}$ and $\boldsymbol{v}$ directions about the $\boldsymbol{w}$ direction. This would be equivalent to a roll, or the rotation of an image about its centre if $\boldsymbol{w}$ were the viewing direction. The second matrix can rotate $\boldsymbol{w}$ to point to any position in the y-z plane as shown in Figure 5(a), where the axis system is viewed from along the positive z axis.



(a) Euler Rotation 2

(b) Euler Rotation 3

Figure 5: Euler Angles

The last rotation about the z axis can make the projection of $\boldsymbol{w}$ point in any direction in the x-y plane as shown in Figure 5(b). Thus the combined transformation can orient $\boldsymbol{w}$ in any direction in the 3D space with the $\boldsymbol{u}$ and $\boldsymbol{v}$ directions aligned in any direction.

Forward Kinematics

Forward kinematics is used in robot control. Given a specification of the Euler angles of each joint or an articulated robot arm we calculate the position and orientation of its end point. This is a well posed problem and can be solved by matrix transformations of space similar to the ones that we have already seen in use - for example in first person shoot 'em ups.

We define a coordinate system at the start of a chain and at each joint as shown in Figure 6. Each new coordinate system is defined in the co-ordinate system of the previous joint. Thus the position $\mathbf{C}^i$ and the direction vectors $[\boldsymbol{u}^i, \boldsymbol{v}^i, \boldsymbol{w}^i]$ of frame i are defined using the frame $i-1$ coordinate system. $\mathbf{C}^i$ and $[\boldsymbol{u}^i, \boldsymbol{v}^i, \boldsymbol{w}^i]$ can be calculated simply using the Euler rotation matrix $\mathcal{R}_E$:

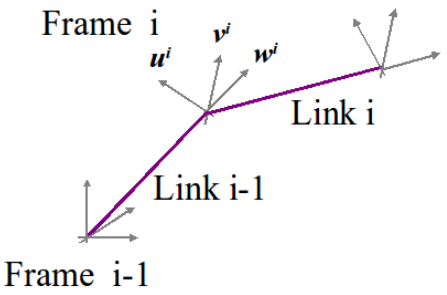


Figure 6: Forward Kinematics

$$\boldsymbol{u}^i = [1,0,0]\mathcal{R}_E \qquad \boldsymbol{v}^i = [0,1,0]\mathcal{R}_E \qquad \boldsymbol{w}^i = [0,0,L_{i-1}]\mathcal{R}_E$$

where L_{i-1} is the length of link $i-1$.

We can express positions and directions in frame i in the coordinate system of frame $i-1$ by a matrix transformation, which is the inverse of the viewing transformation discussed earlier in the course. This can be found by inspection and verified by multiplying out.

$$\begin{bmatrix} u_x & u_y & u_z & 0 \\ v_x & v_y & v_z & 0 \\ w_x & w_y & w_z & 0 \\ C_x & C_y & C_z & 1 \end{bmatrix} \begin{bmatrix} u_x & v_x & w_x & 0 \\ u_y & v_y & w_y & 0 \\ u_z & v_z & w_z & 0 \\ -C \cdot u & -C \cdot v & -C \cdot w & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

To extend this to a whole chain we denote the transformation from frame i to frame $i-1$ as:

$$T_i^{i-1} = \begin{bmatrix} u_x^i & u_y^i & u_z^i & 0 \\ v_x^i & v_y^i & v_z^i & 0 \\ w_x^i & w_y^i & w_z^i & 0 \\ C_x^i & C_y^i & C_z^i & 1 \end{bmatrix}$$

Then the complete forward transformation (using pre-multiplication) can be seen to be:

$$T_n^0 = \prod_{i=n}^{0} T_i^{i-1}$$

Inverse Kinematics

For graphics applications we want the opposite process. We wish to specify some path for the end point of the chain, and calculate the relative positions of all the other links. This is an ill posed problem (there are infinitely many solutions for some chains). Hence we need to find a constrained solution minimising for example, the joint movements.

We would like to use inverse kinematics to calculate in-between frames. For example, to generate ten frames of a leg movement, we define the ten steps that make up the foot movement, and estimate the changes in the Euler angles of the rest of chain that implement those changes. In the simple articulated leg chain there are five Euler angles. The constrained angles are initialised to zero. One way to solve the problem is to use gradient descent.

Let E be the distance between the end point and its target. For each Euler angle θ we find $dE/d\theta$ using forward kinematics, replacing θ with $\theta + \Delta\theta$ and calculating ΔE. We then update the angles using:

$$\theta^t = \theta^{t-1} - \mu \frac{dE}{d\theta}$$

Applying gradient descent with small steps should find a solution that involves only small joint movements. However, over time it is still possible that a skeleton will reach an implausible pose. One approach is to constrain the optimisation to avoid poor poses. Poor pose is difficult to define, and needs analysis of videos of actual motion. This is a current topic for research.

Summary

Currently human intervention is a necessary part of creating realistic living model animation, either using motion capture or expert animator input. Physical modelling and procedural methods are successful in creating movements in inanimate objects. Inverse kinematics can take some of the burden out of model animation simulating living creatures, but is limited and is an interesting current research area.

Tutorial: Warping and Morphing

1. Explain what is meant by the following equation:

   morphing = $(\text{warping})^2$ + blending

2. In the algorithm developed by Beier and Neeley pairs of lines are used to specify the warping. In a concrete example two pairs of lines specify a 2D warping: In the source image the line L_1 starts at (1, 1) and ends at (1, 9). The line L_2 starts at (9, 2) and ends at (9, 8). In the target image the corresponding line L_1' starts at (1, 1) and ends at (1, 9) while L_2' starts at (3, 2) and ends at (9, 2). Calculate where the pixel **p** = (5, 5) in the source image would map to in the target image. Assume that the constants controlling the warping are $a = b = p = 1$.

3. An image with 300 x 175 pixels is warped using a two-dimensional free-form deformation based on linear B-splines defined by a 6 x 6 mesh of control points.

   a. Calculate the spacing between control points in pixels.

   b. Calculate the pixel coordinates for the following B-spline integer lattice coordinates *i, j* and the fractional lattice coordinates *u, v*:

      i. (i, j) = (1, 1) and (u, v) = (0, 0)
      ii. (i, j) = (1, 1) and (u, v) = (0.5, 0.5)
      iii. (i, j) = (1, 3) and (u, v) = (0.75, 0.2857)

   c. Calculate the B-spline integer lattice coordinates i, j and the fractional lattice coordinates u, v for the following pixels:

      i. (x, y) = (120, 140)
      ii. (x, y) = (100, 100)
      iii. (x, y) = (150, 130)

   d. Calculate the new location of a pixel (x,y) = (135, 122.5) after warping. The matrix of control points looks as follows:

(1, 4)	(-3, 7)	(3, 8)	(4, 7)	(0, 1)	(2, 3)
(-3, 7)	(-2, 9)	(2, 7)	(3, 1)	(2, -2)	(2, 2)
(4, 2)	(3, 8)	(2, 1)	(4, 2)	(2, 1)	(3, 1)
(3, 2)	(2, 9)	(-3, 8)	(6, 8)	(3, 4)	(3, 5)
(-1, 3)	(-2, 3)	(1, 3)	(2, 3)	(8, 3)	(-4, 2)
(0, 0)	(-2, 1)	(1, 1)	(-2, 2)	(1, 2)	(0, 0)