Real-Time Rendering in Curved Spaces

Jeff Weeks

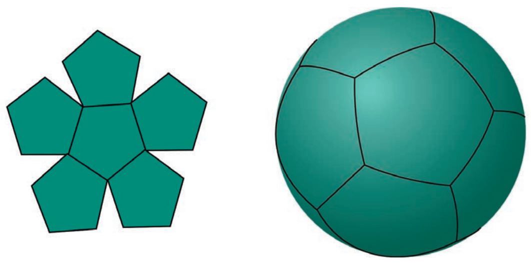
A hypersphere surface provides a finite 3D world in which the user can fly freely without encountering boundaries, while hyperbolic space provides a spacious environment.

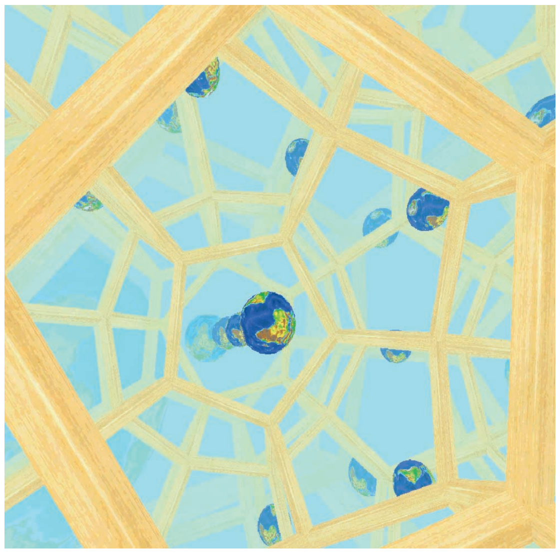
Throughout the 17th and 18th centuries, Euclidean geometry reigned as Western civilization's unquestioned description of physical space. By the early 19th century, however, a nascent interest in higher dimensions led to the realization that an ordinary sphere, which is the 2D surface of a 3D ball, has a higher-dimensional counterpart called the hypersphere, which is the 3D surface of a 4D ball. In the year 1854, Georg Friedrich Bernhard Riemann proposed the hypersphere as a model for the physical universe. Unlike all previous cosmological models, the hypersphere offered a 3D universe with a finite volume but with no troublesome edge or boundary.

Today, the hypersphere model offers new opportunities for 21st century artists and game developers, and for the audiences who enjoy their work. Beyond the appeal of a finite universe, the hypersphere also offers interesting visual effects and rich symmetries not available in flat space models. For example, just as an ordinary sphere allows a tiling by pentagons that would be impossible in a plane (see Figure 1), the hypersphere offers a tiling by dodecahedra that would be impossible in flat space (see Figure 2).

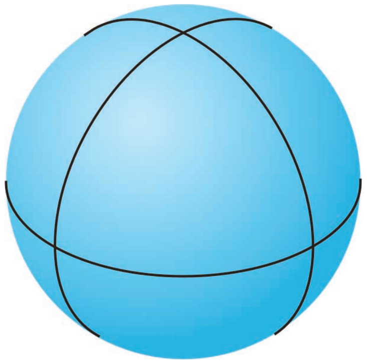
At first glance, you might think that drawing perspectively correct pictures in a hypersphere would be difficult. But the algorithm for rendering a scene in a hypersphere is identical to the standard algorithm for rendering a scene in ordinary flat 3D space. Indeed, the computations are so similar that off-the-shelf 3D graphics cards, when fed the correct matrices, will do real-time animations in a hypersphere just as easily and as quickly as they do in flat space.

The sample code accompanying this article (available from http://www.northnet.org/weeks/CurvedSpaces/) implements curved-space rendering in both OpenGL (http://www.opengl.org) and Direct3D (http://www.microsoft.com/windows/directx/). The sample code can be an easy starting point for readers wishing to explore the possibilities. Even readers not wishing to create their own curved-space applications might enjoy the beauty of the underlying geometry, and students might find that understanding curved-space rendering takes some of the mystery out of the standard flat-space algorithm.

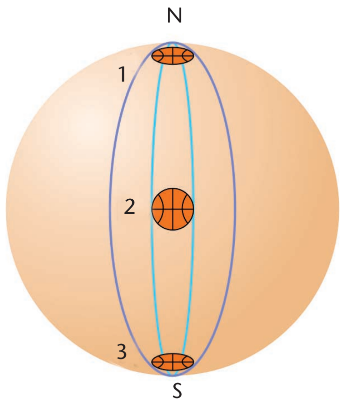

1 Flat pentagons can't tile a plane, but 12 spherical pentagons fit snugly to tile a sphere.


2 Ordinary dodecahedra can't tile flat space, but 120 spherical dodecahedra fit snugly to tile a hypersphere.

0272-1716/02/$17.00 © 2002 IEEE



3 The coordinate planes slice a sphere along three mutually perpendicular circles.



4 The hypersphere's surprising optical properties are best understood by studying optics on the surface of an ordinary sphere. To an observer at the north pole, the basketball, which has been squished onto the sphere's 2D surface, looks large when it's nearby in the (1) northern hemisphere because it subtends a wide angle in the observer's field of view. The same squished basketball looks small when it sits on the (2) equator because it subtends a narrow angle. When the basketball is near the (3) south pole, it again looks large because the convergence of light beams on the sphere causes the basketball again to subtend a wide angle in the observer's field of view. This same optical phenomenon occurs in the hypersphere, causing distant images of the Earth in Figure 2 to appear large.

What's a hypersphere?

The official definition of a hypersphere is simple. Just as the equation $x^2 + y^2 + z^2 = 1$ defines an ordinary sphere as a 2D surface in 3D Euclidean space, the equation $x^2 + y^2 + z^2 + w^2 = 1$ defines a hypersphere as a 3D hypersurface in 4D Euclidean space. A plane intersects an ordinary sphere in the shape of a circle. In particular, the *xy*, *yz*, and *zx* coordinate planes intersect a sphere in a set of three mutually perpendicular great circles (see Figure 3). Similarly, a hyperplane intersects a hypersphere in the shape of a sphere. In particular, the *xyz*, *yzw*, *zwx*, and *wxy* coordinate hyperplanes intersect a hypersphere in a set of four mutually perpendicular great spheres.

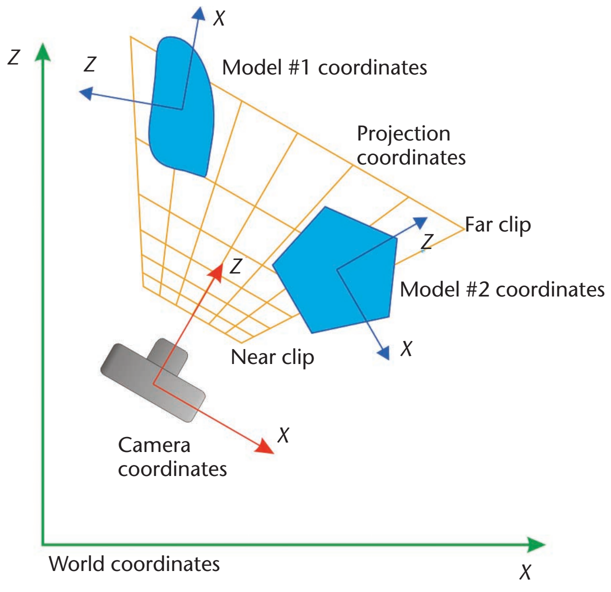
Visualizing the hypersphere is much easier than visualizing the 4D space in which it sits. Remember, the hypersphere is the 3D surface of a 4D ball, not the 4D ball itself, so even though it's a curved space, it's still only 3D. You can understand curved-space rendering—and write your own curved space animations—without ever visualizing four dimensions. Simply do your reasoning using sketches of ordinary spheres in three dimensions, then restore the fourth dimension when it's time to write your code.

The hypersphere has some surprising optical properties. Imagine you're in a hypersphere and you throw a basketball (see Figure 4). At first, the basketball is close to you and it subtends a fairly large angle in your field of view. As the basketball moves away, it subtends a smaller angle. The subtended angle reaches a minimum when the basketball is a quarter of the way around the sphere. As the basketball continues moving away, the subtended angle starts to increase, making the basketball appear larger in your field of view. This phenomenon explains why some distant planets in Figure 2 appear larger than the closer ones.

Flat space rendering

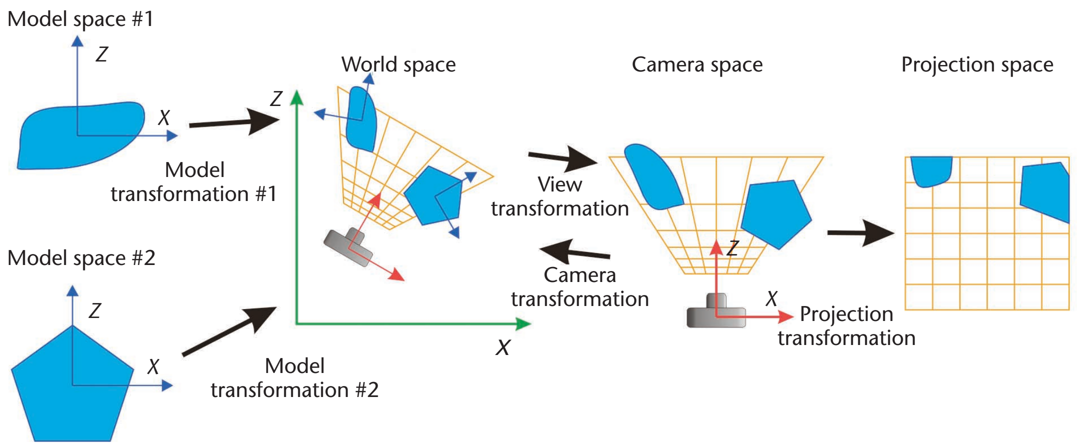
To set a clear starting point and establish terminology, I'll review the standard rendering pipeline[1] and give it a new geometric interpretation. Even though our computations will be 4D, I've drawn the figures one dimension lower by suppressing the *y* coordinate. In all cases the suppressed *y* coordinate behaves just like the *x* coordinate.

Standard 3D graphics uses the following four coordinate systems (see Figure 5):

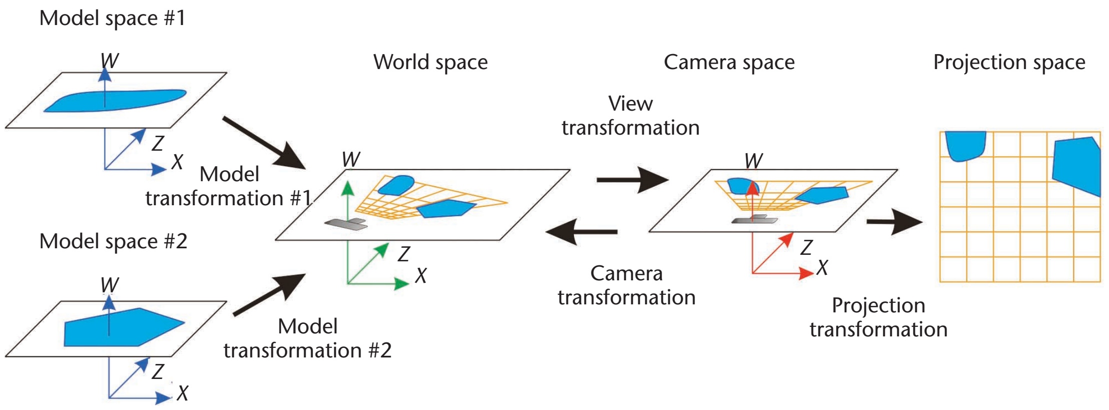


5 The four coordinate systems of standard 3D graphics.

- *World coordinates*. The overall coordinate system that contains everything else.
- *Model coordinates*. Each object in the scene has its own local coordinate system, typically with the object centered at the local origin.
- *Camera coordinates*. The camera sits at the origin of its own coordinate system, looking in the positive *z* direction (Direct3D) or negative *z* direction (OpenGL).



6 You can visualize the four coordinate systems as separate spaces.



7 Each space can be a 3D hyperplane in 4D space so the transformations connecting them can be represented as matrices.

- *Projection coordinates*. The camera's field of view is a pyramid with the camera at the apex. The near and far clipping planes further restrict the field of view to the so-called view frustum. Projection coordinates dictate how the view frustum projects onto the final 2D image. The x and y coordinates give a point's horizontal and vertical coordinates in the final image, while the z coordinate gives its depth.

For clarity, it's convenient to work not just with separate coordinate systems but with separate spaces (see Figure 6). Separating the spaces makes it easier to visualize the transformations. The camera transformation, which places the camera in world space, is the inverse of the view transformation, which places the world in camera space.

The only problem with this scheme is that the transformations, which include translational components, can't be represented by matrices. The standard solution is to add an extra dimension, replacing each 3D point (x, y, z) with the 4D point $(x, y, z, 1)$. Geometrically, we visualize each 3D space as a slice of 4D space at height $w = 1$ (see Figure 7). The model, view, and camera transformations are now linear, and can be encoded as matrices:

$$\begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \begin{pmatrix} \text{rot}_{xx} & \text{rot}_{xy} & \text{rot}_{xz} & \text{trans}_x \\ \text{rot}_{yx} & \text{rot}_{yy} & \text{rot}_{yz} & \text{trans}_y \\ \text{rot}_{zx} & \text{rot}_{zy} & \text{rot}_{zz} & \text{trans}_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

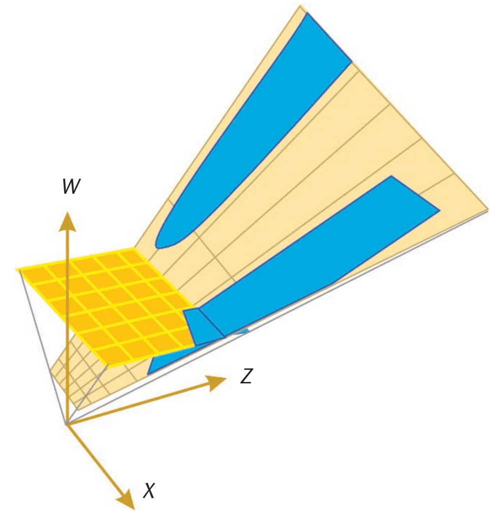
The projection transformation is almost linear, but not quite. A linear matrix multiplication takes the hyperplane at height $w = 1$ in camera space to a slanted hyperplane in 4D projection space (see Figure 8). Numerically, each point $(x, y, z, 1)$ in camera space maps to a point (x', y', z', w') in 4D projection space. We must then divide by the last coordinate to rescale to $(x'/w', y'/w', z'/w', 1)$. Geometrically, rescaling projects the slanted hyperplane shown in Figure 8 onto the horizontal hyperplane at $w = 1$, which provides the desired image of the view frustum as a box in 3D projection space.

Modern 3D graphics cards implement the whole rendering pipeline: They apply the model, view, and projection matrices; clip as necessary; project into the box in 3D projection space; then rasterize the result. Because the whole procedure runs in hardware, detailed scenes can be rendered in real time. Standard 3D graphics cards can render curved space scenes in real time too.

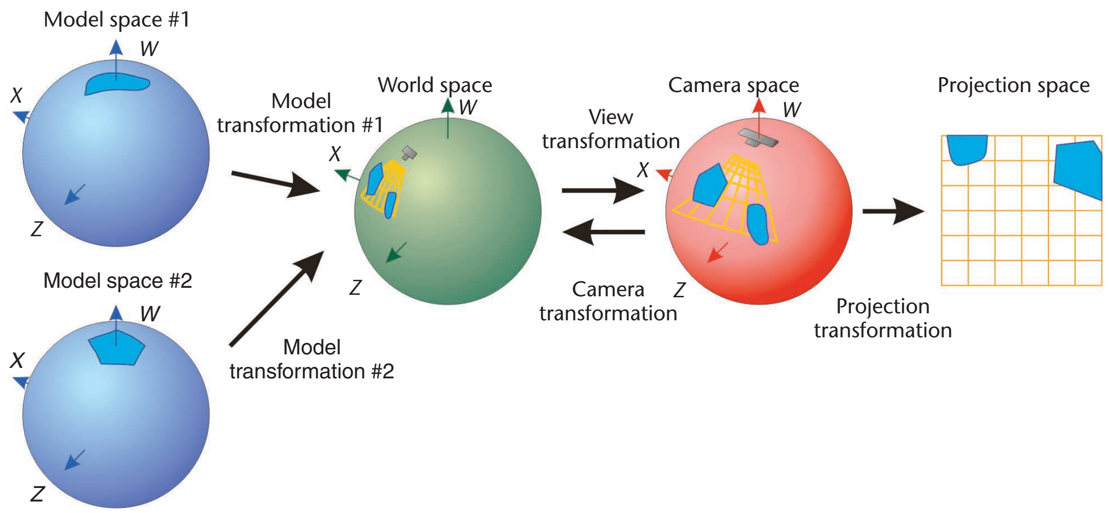
Spherical rendering

Photorealistic curved-space rendering opens a window into a previously unseen world: when you view the computer-generated image in the comfort of your flat-space home or office, you see exactly the same objects in exactly the same directions as you would in the desired curved-space scene, giving you the illusion of living in curved space. To render photorealistic images in a hypersphere, our conceptual imagery changes, but the computational algorithm remains the same. We simply replace the 3D slice at height $w = 1$ with a unit hypersphere (see Figure 9) and proceed as before.

Because the model, world, and camera spaces are now hyperspheres, the transformations relating them are given by pure rotation matrices, unlike the hybrid rotation-translation matrices used in flat-space rendering. For the projection transformation, we can temporarily use the same matrix used in flat space. Spherical rendering uses different model and view matrices, but the algorithm for processing the matrices is identical to the flat-space algorithm, which is true not only in theory but also in practice. Off-the-shelf 3D graphics hardware renders images in spherical space as well as it does in flat space, never noticing that it's processing nonstandard model and view matrices.

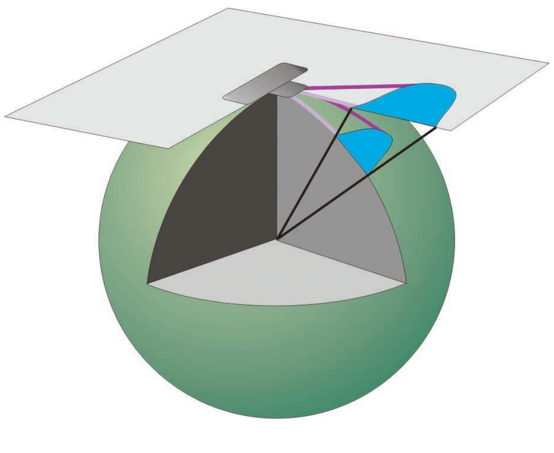


8 The projection transformation matrix takes the view frustum to a slanting hyperplane (the light tan color) in 4D projection space. Dividing by the *w* coordinate normalizes it to the standard box in the hyperplane at height *w* = 1 (the darker orange color).

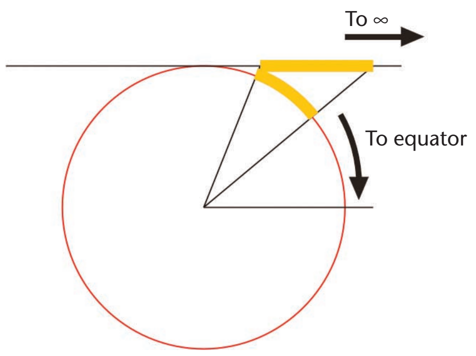


9 The rendering pipeline for a hypersphere is the same as the pipeline for flat space, except that the model, view, and camera transformations are now pure rotations.

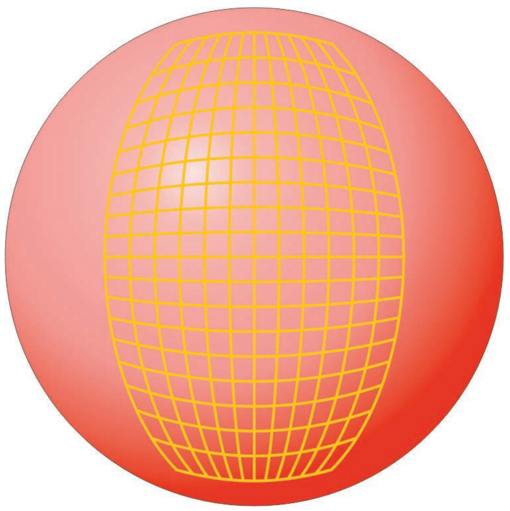
To verify that the flat-space projection matrix gives a perspectively correct image even in spherical rendering, you can simply project each object on the hypersphere radially onto the flat space at height $w = 1$ (see Figure 10). An object at (x, y, z, w) on the hypersphere projects to $(x/w, y/w, z/w, 1)$ in the flat space. Most importantly, the line of sight from the camera to the object—which follows a great circle along the hypersphere—projects to a straight line in the flat space. This proves that a camera viewing the projected objects in the flat space sees exactly the same image that the original camera sees in the hypersphere.



10 If you project each object from the surface of the hypersphere onto the flat space at height *w* = 1, the camera's view in the flat space is then the same as its view on the surface of the hypersphere itself.



11 **Even with the far clipping distance set to infinity, the view frustum's projection onto the hypersphere reaches only to the equator.**



12 **In spherical rendering, a banana-shaped region plays the role of the view frustum. The projection transformation will map this view banana to the clipping box in projection space.**

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & -z_0 \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

OpenGL banana matrix
Camera looks down negative z-axis
Clipping box spans $-w' \le z' \le w'$
Matrix acts as (matrix)(column vector)

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1/2 & 1 \\ 0 & 0 & -z_0/2 & 0 \end{pmatrix}$$

Direct3D banana matrix
Camera looks down positive z-axis
Clipping box spans $0 < z' \le w'$
Matrix acts as (row vector) (matrix)

13 **Banana projection matrix.**

Thus a naive, but correct, algorithm would entail the following steps:

1. Project the scene radially from the hypersphere onto the flat space.
2. Apply the usual flat-space projection matrix to map camera space into 4D projection space.
3. Divide by the last coordinate, as in the flat space algorithm, to arrive at the correct point in 3D projection space.

Surprisingly, we can skip the first step and apply the second and third steps directly to the point (x, y, z, w) on the hypersphere. The proof is as follows: If two vectors (x, y, z, w) and (cx, cy, cz, cw) are scalar multiples of each other, the projection matrix takes them to distinct points (x', y', z', w') and (cx', cy', cz', cw') in 4D projection space. But then dividing each one by its last coordinate takes them to the same point $(x'/w', y'/w', z'/w')$ in 3D projection space. In the special case that $c = 1/w$, steps 2 and 3 take the point (x, y, z, w) on the hypersphere and the point $(x/w, y/w, z/w, 1)$ in the flat space to the same point in 3D projection space.

Now that we've eliminated step 1, the spherical projection algorithm is identical to the standard flat-space algorithm. The algorithm produces a correct image in a hypersphere even when using the standard flat-space projection matrix, but it's needlessly shortsighted. Figure 11 shows that even with the far clipping distance set to infinity, the view frustum on the hypersphere stops at the equator. We get a deeper view into the scene by extending the view frustum to a view banana (see Figure 12).

The banana projection matrix (see Figure 13) gives us a 90 degree field of view beginning at a near clipping plane z_0 radians in front of the camera and ending at a far clipping plane z_0 radians short of the antipode. Strictly speaking, z_0 is the tangent of the clipping distance, but $\tan z_0 \approx z_0$ when z_0 is small, so in practice we ignore this distinction. The banana projection matrix takes slightly different forms for OpenGL and Direct3D to accommodate differing camera, clipping, and notational conventions, but achieves the same final result.

It's easy to check that the banana projection matrix really does take the view banana to the clipping box. For sake of discussion, consider the OpenGL case. The view banana's corners lie at $(1/\sqrt{1+3z_0^2})$ $(\pm z_0, \pm z_0, -z_0, \pm 1)$, but we might safely ignore the $1/\sqrt{1+3z_0^2}$ in front because a constant scale factor is irrelevant. The banana projection matrix takes the banana's near vertices $(\pm z_0, \pm z_0, -z_0, +1)$ to $(\pm z_0, \pm z_0, -z_0, z_0)$ in 4D projection space, which scale to $(\pm 1, \pm 1, -1)$ in 3D projection space. Similarly, the banana's far vertices $(\pm z_0, \pm z_0, -z_0, -1)$ go to $(\pm z_0, \pm z_0, z_0, z_0)$ and thence to $(\pm 1, \pm 1, 1)$. In other words, the banana's corners map precisely to the corners of the clipping box in projection space, as desired.

Optics in the hypersphere differ from optics in flat space. Assume for the sake of discussion that you're at the north pole of a hypersphere. Your lines of sight spread into the hemisphere in front of you, reconverge at the south pole, pass through the south pole and spread out into the hemisphere behind you, and finally reconverge at the north pole, hitting the back of your head. If you teleport yourself into an otherwise empty hypersphere, you see the back of your own head filling the whole sky.

You might expect the standard rendering algorithm to draw objects in the back hemisphere ($z > 0$ in OpenGL, $z < 0$ in Direct3D) along with those in the front hemisphere ($z < 0$ in OpenGL, $z > 0$ in Direct3D). For example, the OpenGL banana projection matrix takes an object at (0, 0, 1, 0) in camera space to the point (0, 0, 0, –1) in 4D projection space, then to (0, 0, 0) in 3D projection space. This point lies at the center of the clipping box, so you might expect it to be drawn at the center of your screen at intermediate depth. In practice, the object fails to appear because OpenGL uses the computationally efficient clipping equations

$$-w' \leq x' \leq w'$$
$$-w' \leq y' \leq w'$$
$$-w' \leq z' \leq w'$$

instead of the more conceptual equations:

$$-1 \leq x'/w' \leq 1$$
$$-1 \leq y'/w' \leq 1$$
$$-1 \leq z'/w' \leq 1$$

The two sets of equations are of course equivalent when w' is positive. But when w' is negative, they yield different results. In the previous example, $(x', y', z', w') = (0, 0, 0, -1)$ satisfies the second set of equations but not the first. In effect, the first set of equations imposes the additional restriction that $w' > 0$ in projection space, which in OpenGL is equivalent to saying that $z < 0$ in camera space. Similar considerations apply in Direct3D, again with the result that only objects in the front hemisphere ($w' > 0$ in projection space or $z > 0$ in camera space) will be rendered.

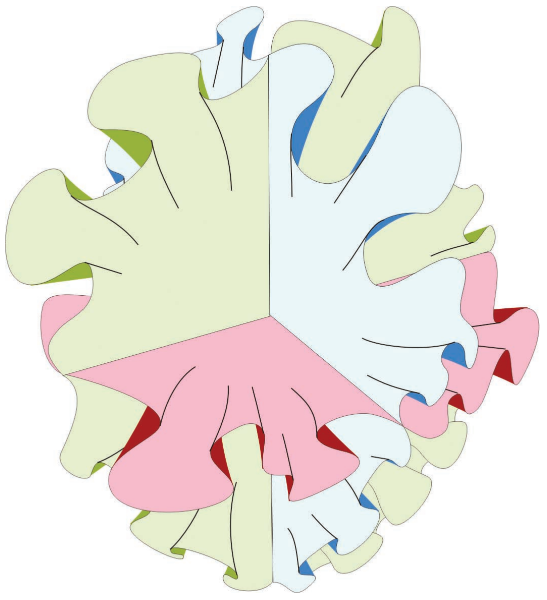
The condition $w' > 0$ turns out to be a blessing in disguise. Without it, objects from the back hemisphere, which span the full range of depth values in projection space ($-1 \leq z'/w' \leq 1$ in OpenGL, $0 \leq z'/w' \leq 1$ in Direct3D), would get intermingled among the objects in the front hemisphere. The $w' > 0$ restriction makes it easy to render the front and back hemispheres separately, one in front of the other.

Two-pass spherical rendering has five steps:

1. Modify the projection matrix to compress the whole view banana into the front half of the clipping box in projection space ($-1 \leq z'/w' \leq 0$ in OpenGL or $0 \leq z'/w' \leq 1/2$ in Direct3D).
2. Render the scene. The $w' > 0$ restriction ensures that only objects in the front hemisphere appear.
3. Move the camera to its antipodal point, facing the back hemisphere.
4. Modify the projection matrix to compress the view banana into the back half of the clipping box ($0 \leq z'/w' \leq 1$ in OpenGL or $1/2 \leq z'/w' \leq 1$ in Direct3D).
5. Rerender the scene. The $w' > 0$ restriction now ensures that only objects in what was originally the back hemisphere appear.

The contents of the front hemisphere end up in the front half of the clipping box in projection space, while the contents of the back hemisphere end up in the back half of the box.

The hypersphere wasn't the only curved space discovered in the 19th century. In the 1820s, Nikolai Ivanovich Lobachevski and János Bolyai independently discovered hyperbolic space. Hyperbolic space is the opposite of spherical space. While spherical space closes up on itself, hyperbolic space opens outward even more widely than flat space.


14 Each slice of hyperbolic space is a hyperbolic plane. The volume enclosed by a sphere grows more quickly than the expected $(4/3)\pi r^3$.

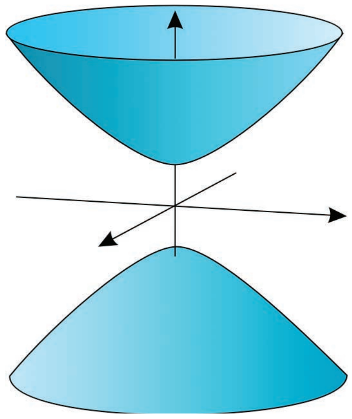
What's hyperbolic space?

Curly leaf lettuce makes an excellent model of the hyperbolic plane. You can make yourself a paper model of the hyperbolic plane by cutting a large number of equilateral triangles from a few sheets of paper and taping the triangles together so that exactly seven triangles (not six) meet at each vertex in the resulting surface. Either way, with lettuce or a paper surface, the hyperbolic plane is roomy in the sense that the area A enclosed by a circle of radius r grows much faster than the expected $A = \pi r^2$. In fact, it quickly approaches an exponential growth rate.

Just as great circles are cross sections of a sphere, and great spheres are cross sections of a hypersphere, 3D hyperbolic space is a space whose cross sections are hyperbolic planes (see Figure 14). Hyperbolic space can't be accurately modeled in ordinary Euclidean space because the volume enclosed by a ball of radius r grows too quickly, so Figure 14 should be considered only a suggestion for imagining hyperbolic space, not a prescription for building a physical model.

Beyond its many applications in math and physics, hyperbolic space has recently found a new application in data visualization.[2] Its rapidly growing volume provides an excellent environment for displaying large data sets. For example, a large binary tree can't be embedded in flat space without extreme crowding because the number of nodes grows exponentially as a function of the depth, but the tree fits comfortably in hyperbolic space with no crowding at all because the volume of hyperbolic space also grows exponentially as a function of the radius.

The mathematics of hyperbolic space are nearly identical to the mathematics of the hypersphere. Indeed, their definitions are the same except for a couple minus signs. A unit sphere, by definition, is the set of points



15 In Euclidean space, the equation $x^2 + y^2 - z^2 = -1$ defines a hyperboloid of two sheets. But in Minkowski space, with distances measured using the Lorentz inner product, each sheet turns out to be a copy of the hyperbolic plane.

that are one unit from the origin. More pedantically, it's the set of vectors $\mathbf{v} = (x, y, z)$ of length $|\mathbf{v}| = 1$. More pedantically still, the length is defined by $|\mathbf{v}|^2 = <\mathbf{v}, \mathbf{v}>$, where the angle brackets denote the so-called inner product, defined as $<\mathbf{u}, \mathbf{v}> = u_x v_x + u_y v_y + u_z v_z$. Stringing these definitions together, you can check that the sphere really does come out to be the set of vectors $\mathbf{v} = (x, y, z)$ satisfying $x^2 + y^2 + z^2 = 1$.

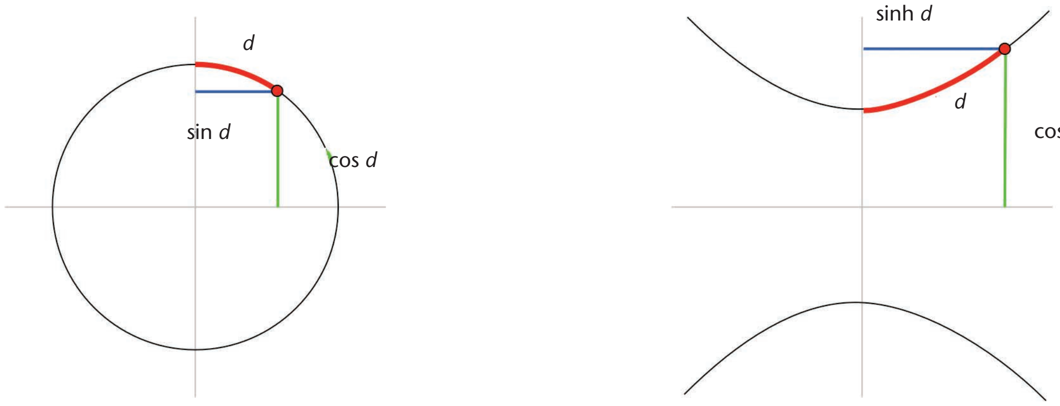
The hyperbolic plane's definition is almost the same. The main difference is that the traditional inner product is replaced with the Lorentz inner product $<\mathbf{u}, \mathbf{v}> = u_x v_x + u_y v_y - u_z v_z$. The only change is the minus sign in front of the $u_z v_z$ term, but that one minus sign makes a world of difference. The squared length $|\mathbf{v}|^2 = <\mathbf{v}, \mathbf{v}>$ of a vector $\mathbf{v}$ doesn't need to be positive. For example, the squared length of the vector (0,0,1) is –1. Euclidean space, with distances measured according to the Lorentz inner product, is known as Minkowski space. The real surprise is that Minkowski space is the geometry of nature, not the geometry of space but the geometry of space-time.

In space-time, the so-called interval between two events $\mathbf{E} = (x, y, z, t)$ and $\mathbf{E}' = (x', y', z', t')$ is given by $|\mathbf{E} - \mathbf{E}'|^2 = <\mathbf{E} - \mathbf{E}', \mathbf{E} - \mathbf{E}'> = (x - x')^2 + (y - y')^2 + (z - z')^2 - (t - t')^2$. When the squared interval is negative, it tells the proper time between two events. When it's positive, it tells the proper distance. And when it's zero, it means the line segment from $\mathbf{E}$ to $\mathbf{E}'$ is the potential path of a light ray. By definition, the proper time is the time as measured by an observer for whom the two events occur in the same location. The proper distance is the distance as measured by an observer for whom the two events take place simultaneously.[3] The interval easily resolves the paradoxes of special relativity.[3,4]

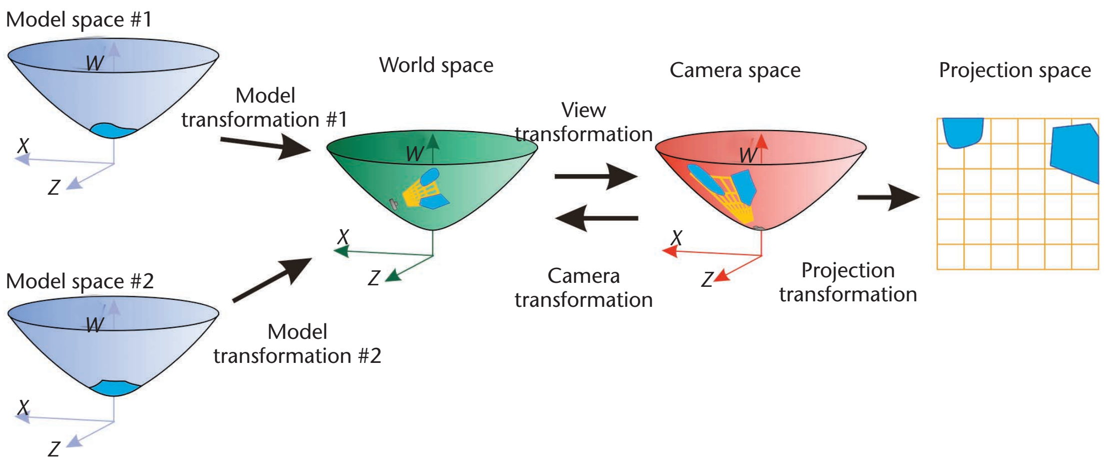
Just as the sphere is most naturally defined as a surface of constant radius in Euclidean space, the hyperbolic plane is most naturally defined as a surface of constant radius in Minkowski space. It's the set of vectors $\mathbf{v}$ satisfying $|\mathbf{v}|^2 = -1$, where $|\mathbf{v}|^2$ is defined using the Lorentz inner product. That is, the hyperbolic plane consists of all vectors $\mathbf{v} = (x, y, z)$ satisfying $|\mathbf{v}|^2 = <\mathbf{v}, \mathbf{v}> = x^2 + y^2 - z^2 = -1$. To our Euclidean eyes, this is the equation of a hyperboloid of two sheets (see Figure 15), but if we measure distances within each sheet using the Lorentz inner product to do the measuring, then we find that each sheet has the geometry of leaf lettuce. The squared length of the segment connecting any two nearby points is always positive, taking us out of the realm of Lorentz distances and back into the realm of ordinary distances. By convention, we take the northern sheet of the hyperboloid ($z > 0$) to be the hyperbolic plane, and ignore the southern sheet ($z < 0$).

We can develop the mathematics of the hyperbolic plane by imitating line-by-line the development of the sphere's mathematics. Just as the spherical functions sin d and cos d are defined to be the horizontal and vertical coordinates of a point d units along a unit circle in the Euclidean plane (see Figure 16), the hyperbolic functions sinh d and cosh d are the horizontal and vertical coordinates of a point d units along a unit hyperbola in the Minkowski plane. In spherical space, the functions sin() and cos() often appear in rotation matrices. For example, the rotation matrix

$$\begin{pmatrix} x' \\ y' \\ z' \\ w' \end{pmatrix} = \begin{pmatrix} \cos d & 0 & 0 & \sin d \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -\sin d & 0 & 0 & \cos d \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix}$$



16 The functions sin d and cos d are, by definition, the coordinates of the point you reach when you travel a Euclidean distance d along a unit circle. Similarly, the functions sinh d and cosh d are, by definition, the coordinates of the point you reach when you travel a Lorentz distance d along a unit hyperbola.



17 The rendering pipeline for hyperbolic space is the same as the pipeline for flat and spherical space, except that the model, view, and camera transformations are now Lorentz transformations.

moves an observer at the north pole (0, 0, 0, 1) a distance *d* along the sphere in the *x* direction. In hyperbolic space the Lorentz matrix

$$\begin{pmatrix} x' \\ y' \\ z' \\ w' \end{pmatrix} = \begin{pmatrix} \cosh d & 0 & 0 & \sinh d \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ \sinh d & 0 & 0 & \cosh d \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix}$$

achieves the same effect.

Hyperbolic rendering

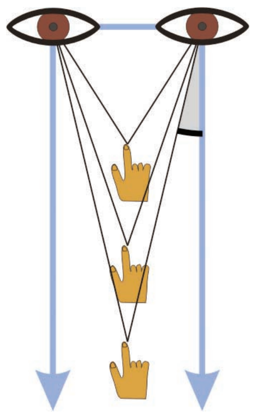
Hyperbolic rendering (see Figure 17) works the same way as flat rendering and spherical rendering, but the model, camera, and view transformations are given by Lorentz matrices.[5] Unlike the spherical case, hyperbolic rendering requires no special care in selecting a projection matrix. Both traditional and banana projection matrices work fine, although the banana matrix leaves the back half of the clipping box in projection space empty.

The proof that this technique yields an accurate image of hyperbolic space is the same as for spherical rendering. Simply project each object on the hyperboloid radially onto the flat space at height $w = 1$. An object at (x, y, z, w) on the hyperboloid projects to $(x/w, y/w, z/w, 1)$ in the flat space, and the line of sight from the camera to the object—which follows a great hyperbola along the hyperboloid—projects to a straight line in the flat space. This technique proves that a camera viewing the projected objects in the flat space sees exactly the same image that the original camera sees in hyperbolic space. As in spherical rendering, the original point (x, y, z, w) and the rescaled point $(x/w, y/w, z/w, 1)$ both map to the same point $(x'/w', y'/w', z'/w')$ in the 3D projection space. The same algorithm that works in flat space and spherical space produces an accurate image in hyperbolic space as well.

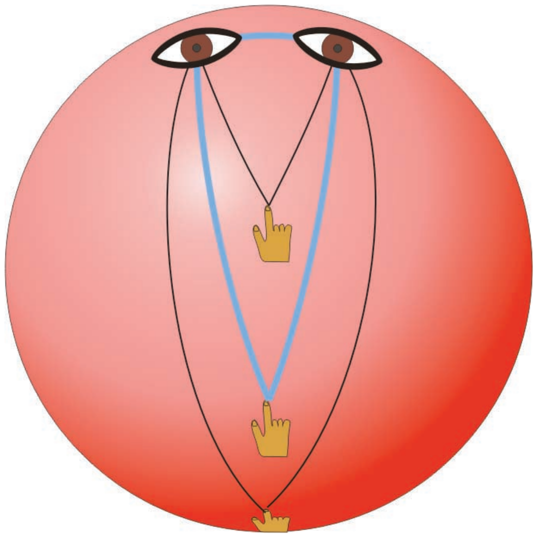
Stereoscopic vision

If you hold your finger 10 cm in front of your face and focus in on it, you'll need to look cross-eyed to see it properly. If you move your finger slowly away from you, the parallax angle—the angle at which your eyes are directed inward—gradually decreases (see Figure 18). In flat space, the parallax angle goes to zero as your finger moves to infinity.

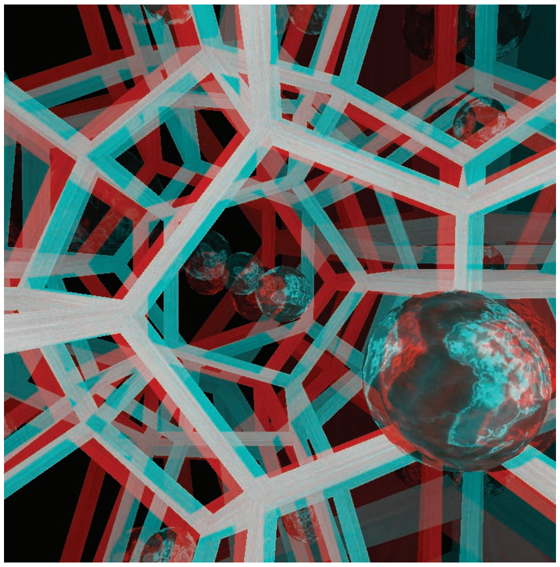
In spherical space, the parallax angle reaches zero for objects that are only a quarter of the way around the sphere (see Figure 19). If each eye looks straight for-



18 As an object recedes from the observer in flat space, the parallax angle θ goes to zero.



19 When an object recedes from the observer in spherical space, the parallax angle reaches zero when the object is π/2 radians away, as illustrated by the blue triangle with right angles at the observer's eyes. As the object moves even farther away, the parallax angle becomes negative.



20 The same scene as in Figure 2, but rendered for viewing with red-blue glasses.

ward, the two lines of sight start off parallel but eventually converge at the object, just as two travelers who start walking due south from different points on the Earth's equator will eventually meet at the south pole. For objects more than a quarter of the way around the sphere, the parallax angle becomes negative. That is, you must direct your eyes outward to focus on the objects.

I initially thought that with negative parallax, the human visual system wouldn't be able to fuse the left- and right-eye views into a single coherent image, and that the user would end up seeing double. Much to my surprise, when using a large screen in a lecture hall with a full 90-degree field of view, most people could easily fuse the two views and see a coherent image. The objects with negative parallax appear to be about 5 or 10 meters away, which is the maximum distance detectable by human stereoscopic vision.

As an object passes through the observer's antipodal point, the parallax angle abruptly switches from extremely negative to extremely positive, and the object suddenly appears very near, although it might nevertheless be dim due to fog effects. From there, the parallax angle again decreases, passing through zero as the object passes three quarters of the way, and remaining negative for the last quarter of its trip. At the finish, the object hits us in the back of our head, or, if we duck, it passes us and we again see it nearby with a large positive parallax.

In hyperbolic space, by contrast, the parallax angle never even approaches zero. For example, if your eyes are 7 cm apart and you are in a standard hyperbolic space (defined in Minkowski space by the equation $|\mathbf{v}|^2 = -1 \text{ meter}^2$), then the parallax angle will always be greater than 2 degrees, even when viewing very distant objects. Long-time residents of hyperbolic space would be accustomed to the fact that a parallax angle of about 2 degrees means that the object is very far away, just as we humans in our approximately flat space subconsciously know that a parallax angle of zero means an object is far. Human tourists to hyperbolic space, however, would be confused. Because the parallax angle always exceeds 2 degrees, they would have the subjective impression that all of hyperbolic space, in spite of its vast roominess, was packed into a finite ball with a radius of 1 meter.

If you own a pair of red-blue glasses, you can experiment with curved space parallax. If you download the sample programs from http://www.northnet.org/weeks/CurvedSpaces, simply run CurvedSpaces-Complete-[GL or D3D].exe, and choose Stereo from the Options menu. The effect is most convincing when you stand in a dark room about 1 meter away from the center of a large screen, with the image filling your whole field of view. But even on an ordinary computer monitor you'll get a satisfactory sensation of depth. Screen shots show the view in a hypersphere tiled by 120 dodecahedra, first in standard mode and then in stereoscopic 3D (see Figure 20).

Conclusion

Astrophysicists already use curved-space graphics as part of their research to determine the shape of the real universe, which is still unknown. In particular, curved-space visualizations have provided new insights into the possible topologies for a spherical universe. These insights simplify the microwave-based detection strategy[6,7] from a six-parameter search to a four-parameter search, reducing the algorithm's runtime from weeks to minutes. These same visualization tools are playing an indispensable educational role in explaining the geometry and topology of space to students. Interactive 3D simulations lead students to an understanding that would not be possible with words alone.

I hope that in the future these tools will find additional application in computer graphics and games. The hypersphere offers gamers the novelty of curved space along with the convenience of a finite yet boundaryless space, while hyperbolic space provides the roominess for working with exponentially complicated data sets.

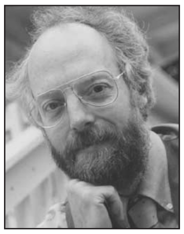
Acknowledgments

I thank Andrew Glassner, Eric Haines, George Hart, Tamara Munzner, and Deej Heath and his students for suggesting improvements to an earlier draft of this article. I also thank the MacArthur Foundation for supporting my work and Charlie Gunn, Stuart Levy, and Mark Phillips for their pioneering work implementing curved space rendering on Unix workstations at the Geometry Center in the late 1980s and early 1990s.[5,8]

References

1. T. Müller and E. Haines, *Real-Time Rendering*, 2nd ed., AK Peters, Natick, Mass., 2002.
2. T. Munzner, "H3: Laying Out Large Directed Graphs in 3D Hyperbolic Space," *Proc. 1997 IEEE Symp. Information Visualization*, IEEE CS Press, Los Alamitos, Calif., 1997, pp. 2-10; http://graphics.stanford.edu/papers/h3.
3. E. Taylor and J. Wheeler, *Spacetime Physics*, W.H. Freeman and Co., San Francisco, 1963.

4. J. Weeks, "The Twin Paradox in a Closed Universe," *American Mathematical Monthly*, vol. 108, 2001, pp. 585-590.
5. M. Phillips and C. Gunn, "Visualizing Hyperbolic Space: Unusual Uses of 4x4 Matrices," *Proc. 1992 Symp. Interactive 3D Graphics*, ACM Press, New York, 1992, pp. 209-214.
6. N. Cornish, D. Spergel and G. Starkman, "Circles in the Sky: Finding Topology with the Microwave Background Radiation," *Classical and Quantum Gravity,* vol. 15, 1998, pp. 2657-2670.
7. J.-P. Luminet, G. Starkman and J. Weeks, "Is Space Finite?" *Scientific American*, April 1999, pp. 90-97.
8. M. Phillips et al., *Geomview: An Interactive 3D Viewing Program for Unix*, http://www.geomview.org.



***Jeff Weeks** is a MacArthur Fellow working in Canton, New York. His research interests include determining the shape of the universe from observational data and developing educational materials for students. He received a PhD in Mathematics from Princeton University.*

Readers may contact Jeff Weeks at 15 Farmer St., Canton NY 13617-1120, email weeks@northnet.org.

For further information on this or any other computing topic, please visit our Digital Library at http://computer.org/publications/dlib.