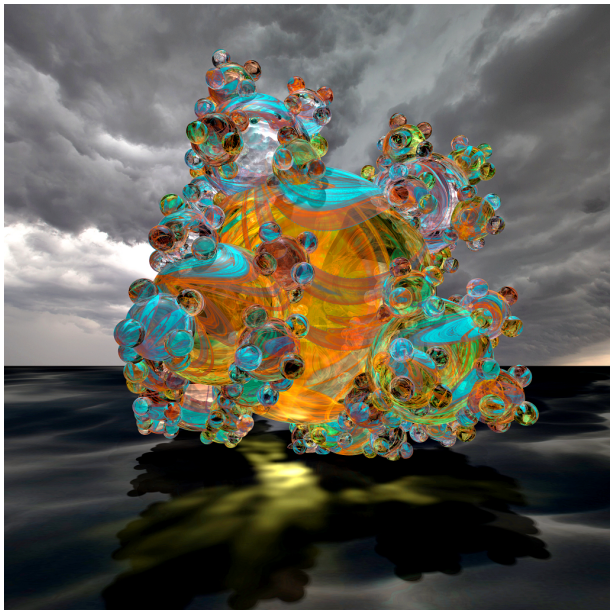
8 Advanced Ray Tracing

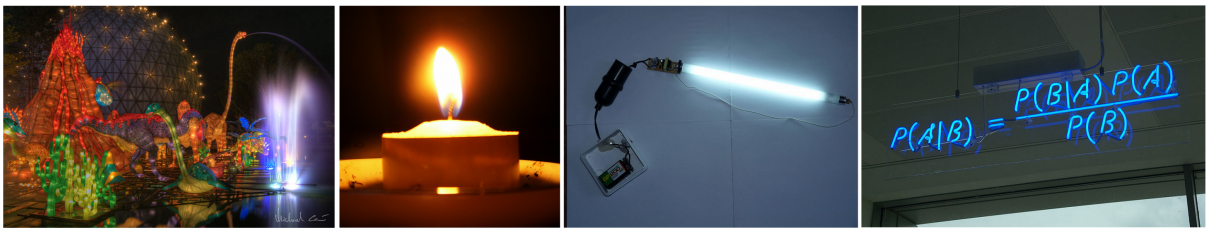
We have now established the fundamental process of ray tracing a scene. Our Whitted ray tracer can render complicated geometry and create realistic shading using a combination of Phong local illumination and limited reflection and refraction for global lighting effects.
Now we will discuss how to extend the Whitted ray tracer to produce more realistic lighting with area light sources, how to smooth out sampling artifacts with antialiasing, using textures to enrich the appearance of objects in the scene, incorporating the thin lens model to produce accurate depth-of-field effects, and using photon mapping for rendering illumination effects due to refraction such as caustics.
Finally, we will take a look into the use of carefully designed data structures such as octrees to accelerate the intersection testing process thus speeding up rendering for scenes with large numbers of objects.



Ray traced scene generated with the Whitted ray tracer plus the advanced rendering techniques discussed in this chapter

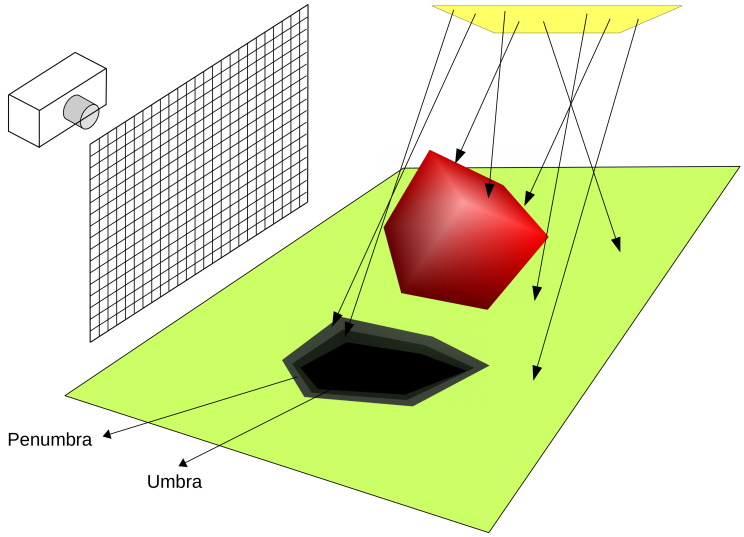
8.1 Area Light Sources and Soft Shadows

Real world light sources do not behave like point light sources. The first obvious difference is that they have a definite shape and size. The shape is three-dimensional but for our purposed here we will assume it is a simple flat shape. Additionally, the light source may cast light rays in a preferred direction rather than in all directions uniformly.



Examples of real world area light sources

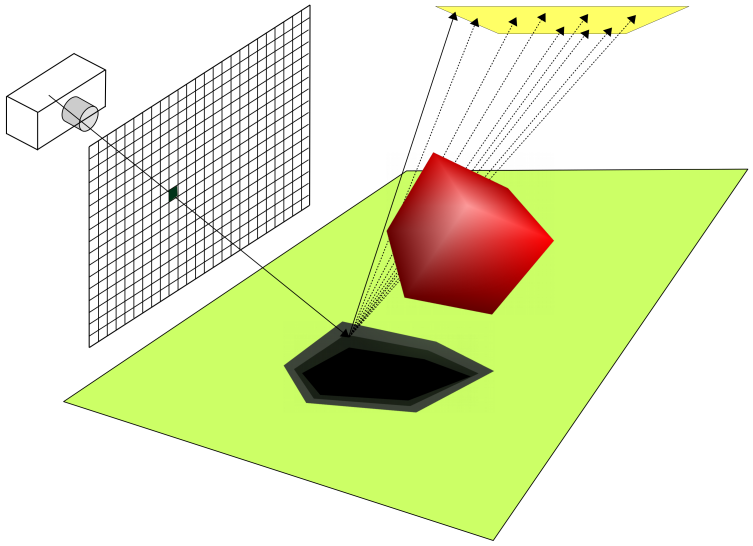
Area light sources produce a variety of visual effects that need to be simulated for rendering realistic images. Most obiously, the light source itself may be visible depending on the point of view of the camera. This is typically handled by introducing an object to the scene that has the right shape, size, and location for the lightsource (e.g. a plane for a typical rectangular fluorescent light), and giving this object the intended colour of the lightsource. The raytracing shading code must know to give this object a uniform colour independent of viewing angle.
Secondly, area light sources produce soft shadows. This means that the transition between the illuminated part of a surface, and the dark part corresponding to the shadow of some object blocking the lightsource is not sharp, but rather darkens gradually, as the object blocks more and more of the lightsource from reaching the surface.



Area light source producing soft shadows. Part of the surface will be completely dark since the entire light source is blocked, but close to the edge of the shaded area, larger and larger portions of the lightsource become visible creating a smooth transition between the fully dark and the fully illuminated regions of the surface. [Sources: Flickr, Wikipedia, Authors: Michael Caven, Gisela Giardino, Dmitry G, Matt Buck]

Implementing soft shadows, as it turns out, is straighforward. Recall that shadow testing in the Whitted ray tracer consists of shooting a ray from the intersection point in the direction of the light

source, and if this ray is blocked, zeroing-out the diffuse and specular components of the Phong local model for that surface point.
With area light sources the process is the same, however, instead of shooting a single shadow ray in the direction of the lightsource, we shoot K rays from the intersection point toward *randomly and uniformly sampled* points on the area lightsource. We count the number of light rays k that are *not* blocked, and then evaluate the reflectance at the intersection as
$E = r_a I_a + \frac{k}{K} r_d I_d \max(0, \vec{n} \cdot \vec{s}) + \frac{k}{K} r_s I_s \max(0, \vec{c} \cdot \vec{m})^\alpha + r_g I_{spec}$
As before, if multiple area light sources are present, the process must be carried out for each area light source and their individual contributions are added up to obtain the final reflectance at the surface.

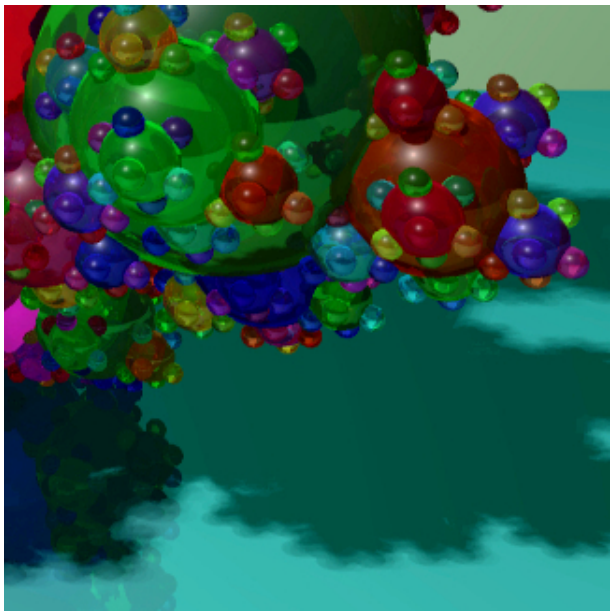


Casting shadow rays toward randomply sampled points on the area lightsource to determine how much light reaches the surface.

```
Sampling area light sources

Determine the percentage of the light source's area visible from
a given surface point

  1.1) Set a counter of unblocked shadow rays k=0
  1.2) Repeat for K shadow rays

       1.2.1) Randomply and uniformly sample a location on the light
              source
```



```
      1.2.2) Cast a shadow ray from the surface point toward the
             sampled location, and check whether it is blocked by
             an object in the scene.
      1.2.3) If the ray is not blocked, increment k=k+1

  1.3) The estimated area of the light source visible from the surface
       point is k/K
```



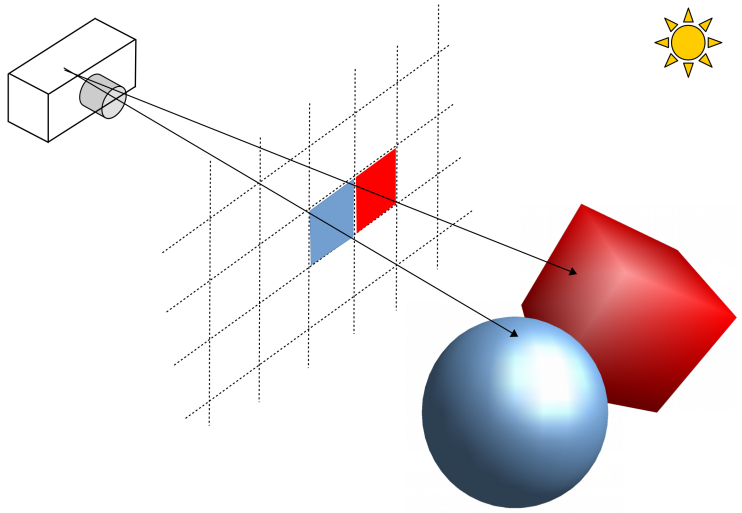
Detail from an image containing an area light source. Notice the soft boundaries of the shadow on the ground plane

Note:
To sample points from an area lightsource that is defined as the result of applying an affine transformation to a canonical object such as a plane, obtain a randomly, uniformly sampled surface point on the canonical object, then apply to it the same transformation used to define the lightsource.

8.2 Antialiasing

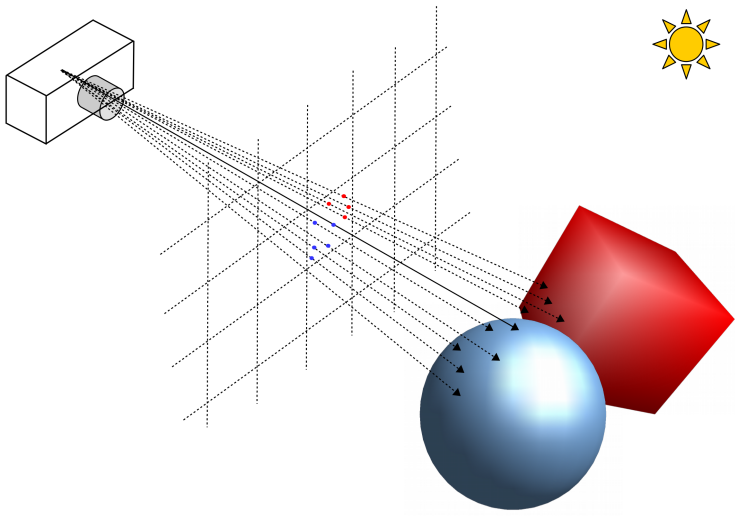
Since the image has a finite resolution, the raytracer will suffer from sampling artifacts around surface boundaries, and along regions where texture or illumination patterns change rapidly. The

resulting *jaggies* are the result of a process called *aliasing*. Aliasing is the result of high-frequency brightness information that can not be captured at the current resolution being folded onto (i.e. aliased) lower frequency brightness changes present in the scene.



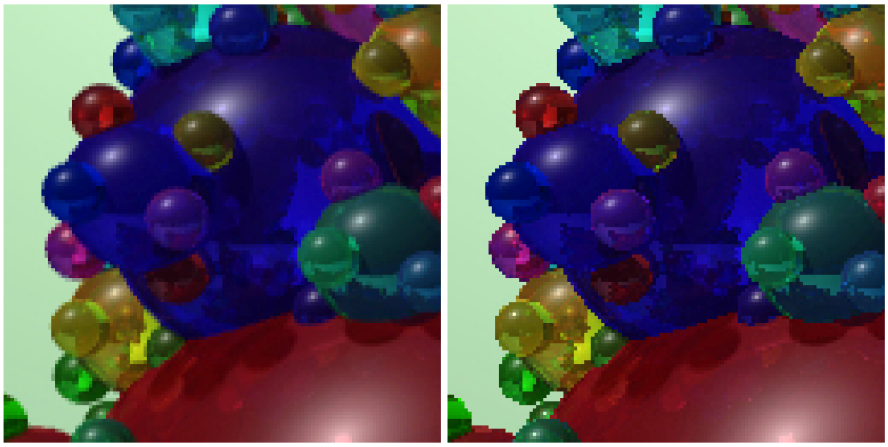
Zoom-in of a small area of the image. Because of the finite resolution, rays through neighbouring pixels may end up hitting completely different objects. The resolution is not sufficient to resolve detail at the boundary and jaggies will be visible in the image

Since aliasing occurs because of limits in the sampling of light coming from the scene, the solution is to *supersample* light rays through image pixels. That is, instead of casting a single ray per pixel (through the centre of the pixel), we randomly and uniformly sample a small number of rays through coordinates within the pixel's area. The colour values returned for these rays are then averaged to obtain the final colour at the pixel.



Casting multiple rays through a pixel, at coordinates randomly and uniformly sampled over the pixel's area, results in a range of colour values that better capture the light arriving from the scene at that pixel. The final pixel colour is the average of the colour values returned for the set of supersampled rays. The resulting image will be smooth and free of jaggies

Note that the supersampling process does not increase the visible resolution of the scene. It only reduces aliasing artifacts. The highest quality scene will have an intrinsically high resolution, as well as anti-aliasing.



Comparison of images rendered with (left) and without (right) antialiasing. Note the smooth boundaries in the image processed with anti-aliasing. Conversely, the image rendered without anti-aliasing shows sharp colour transitions at the boundaries - *jaggies*

8.3 Texture Mapping

In order to achieve realistic results for complex materials, we need to be able to define a surface reflectance that is not uniform over the entire surface of the objects we have in our scene. Complex materials are characterized by surface irregularities and changes in reflectance (colour) across their surface. Simulating these factors accurately is difficult and computationally expensive. However, we can achieve realistic results by using texture mapping - a techinique that allows us to specify the reflectance properties of an object from a separate texture map in such a way that the surface appears to show complex changes in reflectance.
The two natural sources of visual texture on object surfaces are:

- Surface markings — variations in *albedo* (i.e. the total light reflected from ambient and diffuse components of reflection), and
- Surface relief — variations in 3D shape which introduces local variability in shading.

The first aspect of visual texture, changes in abedo, are the target of traditional texture mapping. Later on we will see how the same technique can be applied to simulate small variations in surface relief via the related technique of bump mapping.

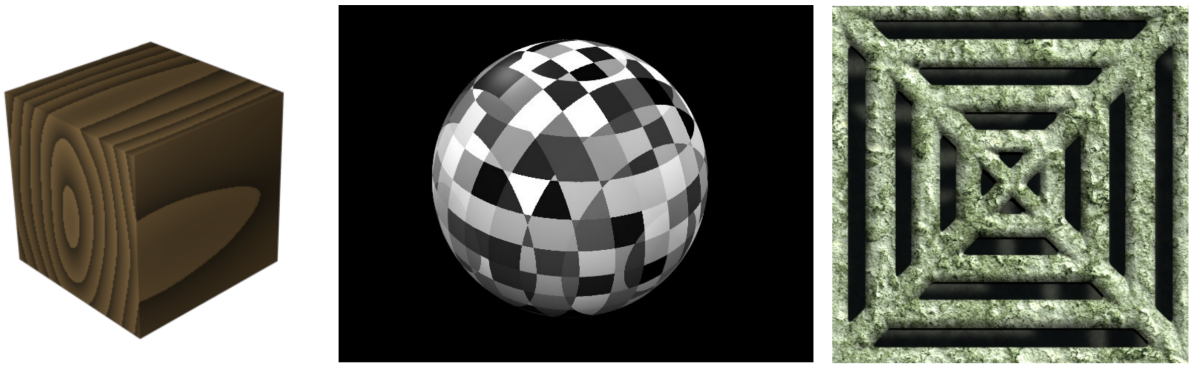
Example of visual texture from changes in surface albedo (leftmost 2 images), and from variations in surface relief (rightmost 2 images). [Source: Wikipedia, Flickr, Authors: Pentocelo, Mitch Featherston, Yann Caradec, Eddi Van W]

8.3.1 Texture Sources

The two most common types of textures used in computer graphics are *procedural textures* and *digital images*.

8.3.2 Texture Procedures

Textures may be defined procedurally, that is, there is a function or set of functions in the program code whose job is to determine the texture colour corresponding to each point along an object's surface. The texture procedure typically involves a stochastic (random) component.
The texture procedure takes as input a point on the object's surface, it returns the reflectance properties at the point: Albedo, and possibly specular and/or alpha values at that point. Examples of procedural textures include checkerboards, fractals, voronid or similar tesselations, and various forms of noise.



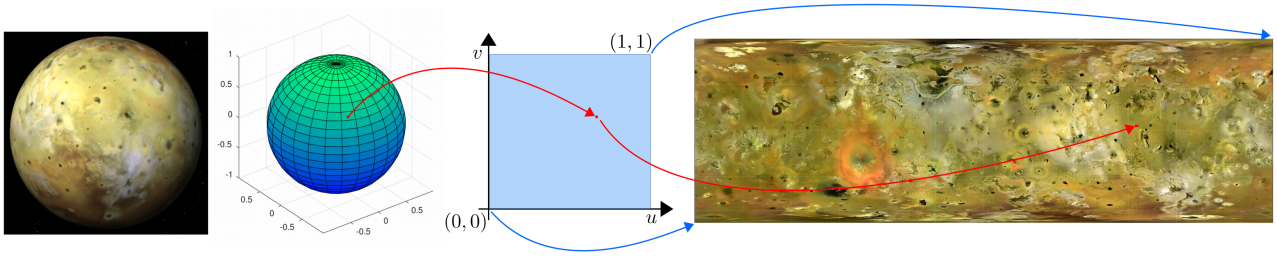
Samples of textures generated procedurally. [Source: Wikipedia, Authors: Falstaff, Soylent Green, Wiksaidit]

8.3.3 Digital Images

We can map any digital image onto a surface. This means we can approximate the appearance of any material for which we have a photograph. The texture image is applied to the object's surface in much the same way as we would apply a decal to a real object.
We can define texture coordinates (u, v) to be in $[0, 1]$. This is done to establish a consistent texture coordinate space that is independent of the texture image's resolution. Each point $[u_0, v_0]$ in texture space is mapped to a corresponding point (x_0, y_0) in the image as $x_0 = u_0 * (s_x - 1)$, $y_0 = v_0 * (s_y - 1)$, where s_x, s_y is the image resolution.
To complete the texture mapping process, we need to establish a correspondance between surface points and pixels in the texture image. To do this, we define a correspondence between object surface coordinates and texture coordinates. For objects consisting of a triangle mesh, each vertex $\bar{p}_i$ in the mesh is associated with a specific texture coordinate (u_i, v_i). For points within each triangle, texture coordinates are interpolated from vertex texture coordinates.
For parametric surfaces, we define a mapping between the two parameters defining the span of the surface and the two texture coordinates. This mapping is continuous, and gives a corresponding texture coordinate to every possible point on the object's surface.

Example:
For a planar patch $\bar{s}(\alpha, \beta) = \bar{p}_0 + \alpha\vec{a} + \beta\vec{b}$, where $0 \leq \alpha \leq 1$ and $0 \leq \beta \leq 1$.
We can define the mapping from object surface points to texture coordinates as $u = \alpha$ and $v = \beta$.

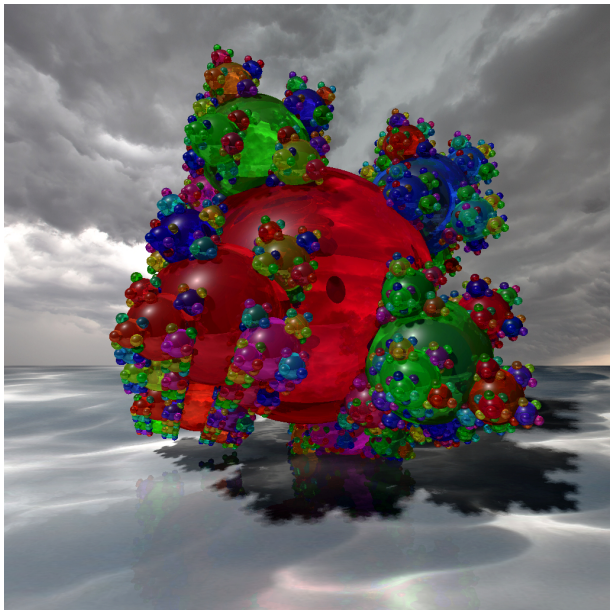
Example:
For a surface of revolution, $\bar{s}(\alpha, \beta) = (c_x(\alpha)\cos(\beta), c_x(\alpha)\sin(\beta), c_z(\alpha))$, with $0 \leq \alpha \leq 1$ and $0 \leq \beta \leq 2\pi$.
The mapping from surface coordinates to texture coordinates can be defined as $u = \alpha$ and $v = \beta/2\pi$.



Texture mapping Jupiter's moon Io. Given the parametric coordinates of a point on the sphere, we map this point onto texture space (u, v), then use the texture coordinates to look up the corresponding pixel in the texture image. [Texture source: NASA]

8.3.4 Textures and Phong Reflectance

Whatever the texture mapping process, the end result is a colour value for the corresponding surface point. This colour value has to affect the ambient and diffuse components of the Phong model. The Phong reflectance at the intersection point becomes:
$E = r_a c_t I_a + r_d c_t I_d \max(0, \vec{n} \cdot \vec{s}) + r_s I_s \max(0, \vec{c} \cdot \vec{m})^\alpha + r_g I_{spec}$,
where c_t is the texture colour. Of course, in RGB the reflectance will have three colour components. Note we could also choose to multiply the specular component, and/or the global reflection component by the texture colour depending on the desired visual behaviour of the surface. For objects with transparency, the texture colour would modulate the colour of the ray transmitted through the object.



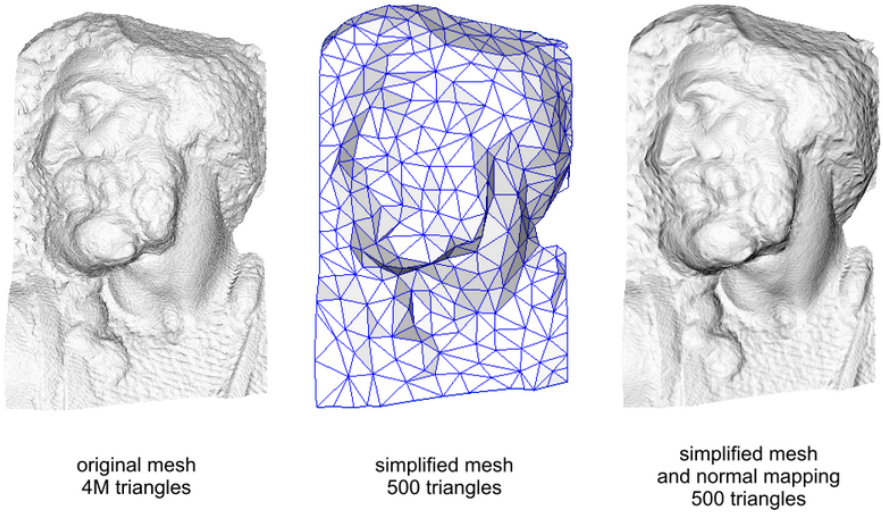
Raytracing texture mapped surfaces - the texture image provides colour information used to modulate the components of the Phong model

```
Texture Mapping

  1.1) Obtain intersection point between the ray and the surface

  1.2) Compute the values of the parametric surface coordinates
        for the corresponding 3D surface

  1.3) Obtain the corresponding texture coordinates

  1.4) Determine the corresponding image coordinates

       1.4.1) Obtain the corresponding colour from the texture
              image. This will involve bi-linear interpolation
              when image coordinates are non-integer
```

8.4 Bump Mapping

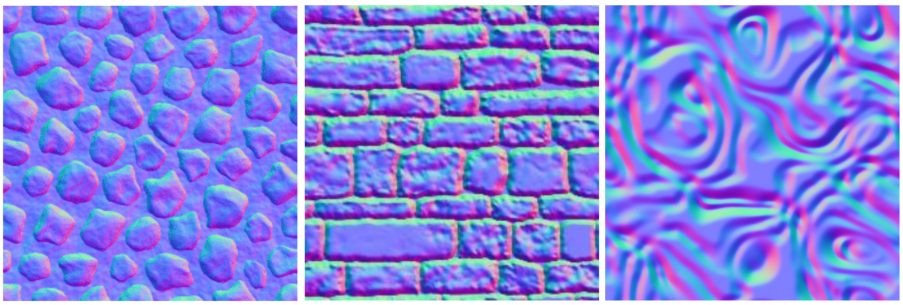
As noted above, the second natural source of visual texture is variations in surface relief. While we could in theory model in great detail the smallest changes in surface geometry for a given object,

this will result in an unmanageable increase in the complexity of the scene and therefore in the time required to render an image. However, we can apply the same process used above to generate textured surfaces to simulate small changes in relief on otherwise simple surfaces so as to increase the visual quality of the rendered scene.



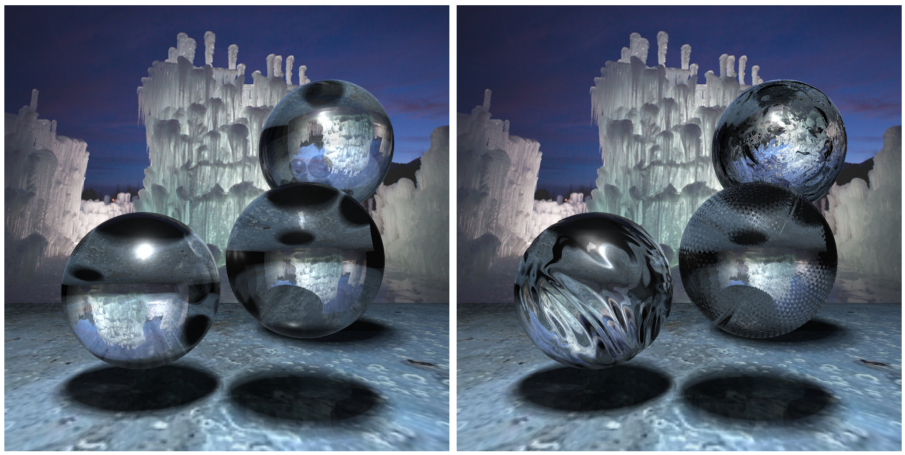
We can use bump mapping to simulate detail-rich surface geometry while keeping the complexity of the object models under control. [Source: Wikipedia, Author: Paolo Cignoni]

There are two common forms of bump mapping. The original bump mapping formulation used a *displacement map* that stored for each point in the surface a small displacement vector. At rendering time, the surface geometry would be modified according to the displacement map, and appropriate surface normals and intersection points would be computed.
However, at the present the most common form of bump mapping is implemented via *normal maps*. A normal map is a special kind of texture image that stores normal vector components for each point on the object's surface. The RGB values in $[0255]$ are used to represent vector components in $[01]$. The normal vector can be encoded either directly (i.e. the correct normal vector for each surface point is stored in the map), or as a perturbation on the surface normal (i.e. we need to combine the information in the normal map with the surface normal to determine the actual surface orientation).



Examples of normal maps. Note that the map itself conveys the impression of changes in surface relief

At rendering time, the normal vector at an intersection point is obtained from the normal map using the same process described above for texture mapping.



Comparison of surface appearance without (left) and with (right) bump mapping. Notice the way the normal map affects refraction, creating a definite impression of surface relief despite the fact that the underlying geometry is still perfectly spherical. Note the limitations of bump-mapping - the object's shadow remains perfectly elliptical

It is worth noting that we can generate map for other visual properties of objects. Specular maps and alpha maps are common, and they are implemented in the same way.

8.5 Photon Mapping

At this point our raytracer is capable of rendering a realistic scene with complex visual surface effects. However, it has one major limitation. The handling of light from secondary bounces is limited to the global specular component of the Whitted ray tracer. This means that certain visual features of images are missing in our renders. Note the shadows of the transparent spheres in

the bump mapping examples above. The shadows are *completely dark*. We would expect these transparent spheres to transmit and possibly concentrate the light from the area light source in some way over the ground plane. However, due to the way our ray tracer handles shadow testing, any points on the scene for which the lightsource is blocked will appear dark. The ray tracer has no way to estimate where refracted light goes and how bright it should look.
The core of the problem is that our Whitted ray tracer is really a crude approximation to actual light transport. We will address the issue of accurately simulating global light transport in the following chapters. For now, we will discuss a common technique for handling refracted and reflected light with our Whitted ray tracer.

Refracting and reflecting objects concentrate light in bright regions known as *caustics*. We use photon mapping to simulate these light effects in simple ray tracing.

Photon mapping handles indirect illumination from reflecting and refracting surfaces by performing one pass of *forward light tracing* before the raytracing process begins. In the previous chapter we noted that forward ray tracing is able to create complex global illumination effects but does so at the cost of a huge computational expense. In photon mapping, we carry out a limited amount of forward light tracing. Since we do not need to generate a smooth image, we can get away with a manageable amount of computation while at the same time allowing our raytracer to render high quality caustics.

8.5.1 Forward Pass

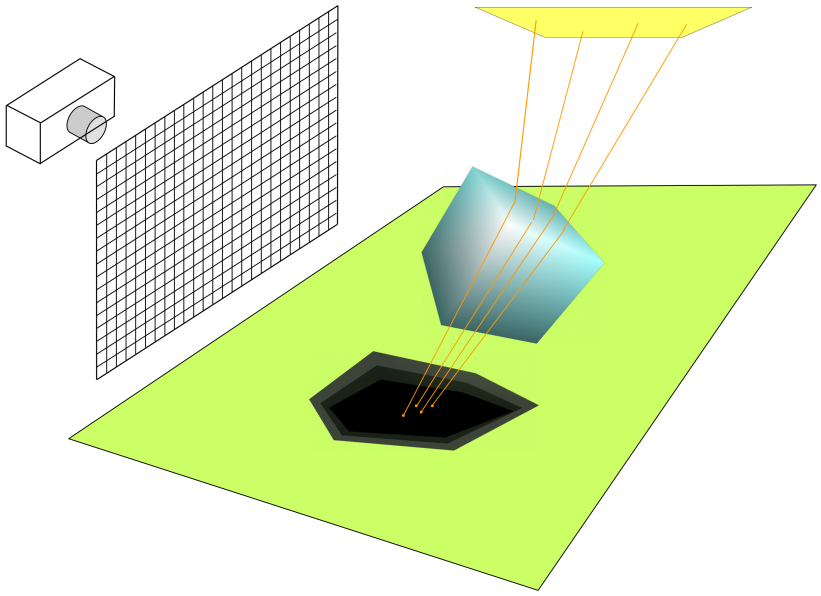
The forward pass works in the following way:

```
Photon Mapping Forward Pass

  1) For each of N light rays

    1.1) Select an area light source. Cast a ray from a random
         location on the lightsource in a random direction.
```



```
1.2) Trace the light ray until

    1.2.1) The ray hits nothing (it leaves the scene). Stop
           tracing

    1.2.2) The ray hits a reflecting/refracting object. Compute
           the direction of the resulting ray and trace this
           ray

    1.2.3) The ray hits a diffuse surface. If the ray has been
           reflected or refracted at least once, store
           one photon at the location of the intersection point
           and stop tracing

    1.2.4) Maximum recursion depth is reached. Stop tracing
```

Simply put, the forward pass traces the path of light rays as they travel through the scene being reflected or refracted by objects and until they hit a diffuse (non reflective, non refractive) surface. At that point, we store a small amount of light, *a photon* in a suitable data structure. The photon contains information about the colour of the ray that hit the surface as well as the location on the surface where the ray hit.

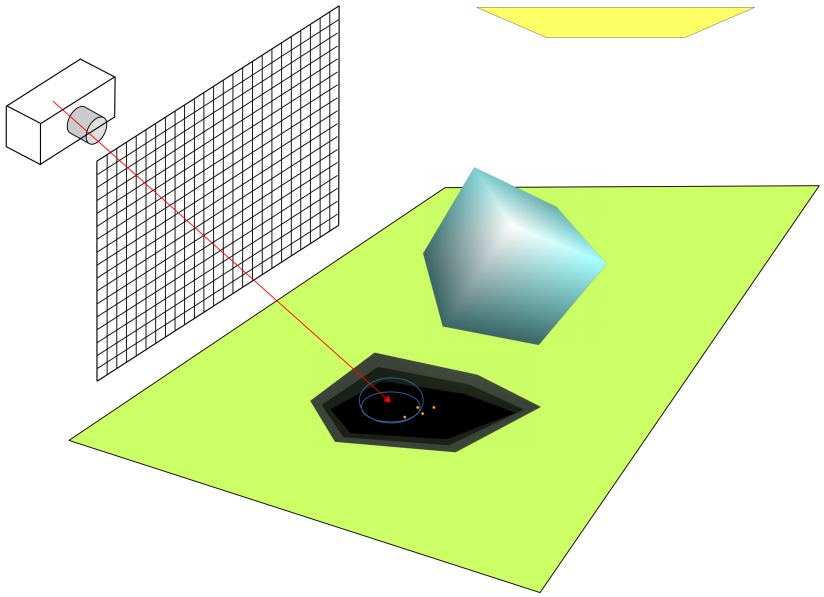


Forward ray tracing through a refractive object. Light rays are bent toward the surface, and at the point where they hit we store a photon.

Note that photons are only stored if the ray has been reflected or refracted at least once. We do not store photons for direct illumination, since direct illumination is handled by the Phong model. Also, note that for forward ray tracing, we must keep track of the colour of the light ray as it is reflected or refracted by scene objects. The initial ray has the same colour as the light source, but its colour will be modulated by the objects it interacts with.

8.5.2 Rendering Pass

Once the forward pass has been completed, we perform a rendering pass that consists of our standard Whitted ray tracer with one modification:
Wherever a ray hits a diffuse surface where photons may have been deposited we do a lookup into our stored photon map and add up the colour contribution of any photons within a sphere centered at the intersection point. The radius of the sphere controls how sharp or smooth the caustics will look - a larger sphere pools brightness from a larger region, and will produce smoother caustics with fewer forward light rays. Sharper caustics will require larger amounts of photons being cast during the forward pass.
In either way, once we have accumulated nearby photons and have obtained an average colour for them, we add their contribution to the radiance at the intersection point:
$E = r_a c_t I_a + r_d c_t I_d \max(0, \vec{n} \cdot \vec{s}) + r_s I_s \max(0, \vec{c} \cdot \vec{m})^\alpha + r_g I_{spec} + \tau k/N I_{photon}$,
where τ is a tnable constant that determines how much brightness is contributed by photons to the radiance at the intersection point, k/N is the proportion of the total number of photons cast that is pooled around the intersection point, and I_{photon} is the average RGB colour of the accumulated photons.



Rendering pass. When a ray hits a diffuse surface, we look up any photons within a small radius of the intersection point, obtain their average colour, and add a small amount of brightness with this colour to the radiance at the intersection point computed by our ray tracing shading model.

8.5.3 Baked-in Radiance

The process described above has the advantage that we do not need the cached photons to create a smooth pattern on diffuse surfaces. The use of a small spherical region for accumulating photon radiance has the effect of smoothing the resulting caustics. However, the downside is that at each intersection point we must perform a look up for nearest neighbours within the data structure that holds the cached photon map. This can become computationally expensive.

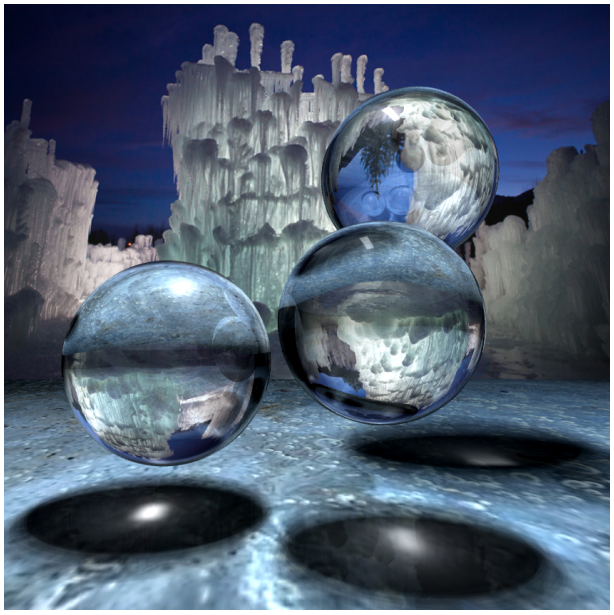
An alternative way to store photon maps that avoids this lookup step is to *bake them* onto initially black texture images. The process is simple - a photon map texture image is associated with every diffuse object in the scene. Each photon map texture is initially black. During the forward pass, if a photon lands on a diffuse surface, we store a small amount of brightness with the corresponding photon's colour at the pixel location in the photon map corresponding to the place where the ray hit the surface (this is done using the same procedure described above for obtaining a texture colour for a surface point).

Once all the photons have been cast and accumulated, each photon map texture is post-processed to smooth out the brightness pattern produced by the cached photons (otherwise the photon map will look like a noisy collection of bright dots on a black background), and to modulate the brightness of the final caustics in the rendered scene.

During the forward rendering pass, at each intersection point for diffuse objects we perform an additional texture lookup and retrieve the amount of light contributed by the photon map at that location, then simply add it to the remaining components of the shading model.

This method for handling photon maps results in a faster rendering pass, at the cost of requiring a larger amount of traced photons to produce smooth caustics. Tuning the post-processing of photon map textures so the final rendered caustics look realistic can be tricky.

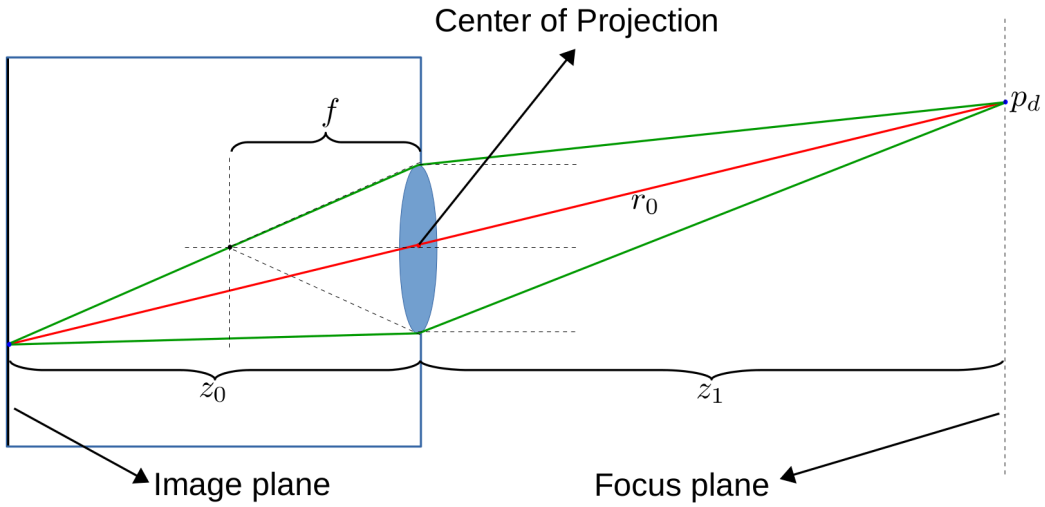
Regardless of the method used, photon mapping can produce stunning results when carefully implemented.



Transparent spheres rendered using photon mapping (baked-in radiance photon map). The caustics produced by refracted light are cleanly rendered.

8.6 Depht of Field

So far we have been working under the assumption of a pinhole camera. As we have seen in previous chapters, real cameras use lenses to focus light onto the imaging plane, thereby allowing for images to be captured in a practical amount of time. Introducing a lens fundamentally changes the way a captured image looks: Certain parts of the image will be in focus, and appear in sharp detail, while out of focus regions will appear blurred. The relative depth interval within the image for which objects appear in focus is called the depth-of-field (DoF).
For a raytraced image to look convincingly like it was captured by a camera, we need to simulate the depth-of-field effect. We can do this using the thin lens model. Recall that the thin lens model replaces the pinhole by an aperture fitted with a thin lens which focuses light according to the thin lens equation:
$\frac{1}{z_0} + \frac{1}{z_1} = \frac{1}{f}$



Thin-lens model. The camera's lens focuses light rays parallel to the optical axis at a distance f behind the aperture. Scene points that will be in-focus are located at a distance x_1 along the optical axis, and will focus onto the image plane at a distance of x_0 behind the aperture. The thin-lens equation gives the relationship between f, x_0, and x_1.

Simulating DoF requires us to replace the single ray through pixel (i, j) by a cone of rays from the aperture all of which converge at a point in space located on the focus plane. The process is straightforward:

```
Depth-of-Field for a pixel at (i,j)

  Inputs: focal length f, aperture size, focus plane distance x_1

  1.0) Compute image plane distance x_0 from the thin-lens equation

  1.1) Cast a ray r_0 through the pixel at (i,j) and the center of
       projection (this is the same ray we trace for a pinhole
       camera). Note that this time the image plane is at distance
       x_0, not f as was the case for the pinhole camera

  1.2) Compute the intersection point p_d of r_0 with the perfect
       focus plane at distance x_1 along the optical axis.

  1.3) For N depth-of-field rays

         1.3.1) Randomly select a point p_0 on the aperture (we can
                approximate the aperture shape with a circle of the
```



```
            specified diameter)

        1.3.1) Cast a ray r_i from p_0 toward the intersection at
               p_d

        1.3.2) Raytrace r_i and accumulate the returned colour onto
               the pixel's colour at (i,j)

  1.4) Divide the values at pixel (i,j) by N to obtain the final
       colour
```

Detail of an image rendered with depth-of-field showing sharp in-focus areas as well as soft focus regions away from the focus plane.

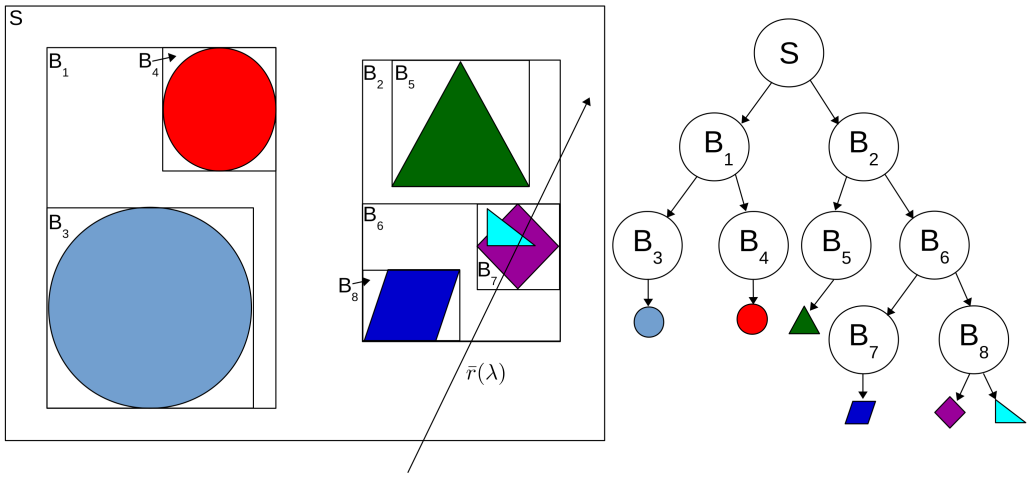
8.7 Accelerated Intersection Testing

So far we have not considered the computational expense of ray tracing. The two basic operations - ray casting, and intersection testing, will easily be repeated billions of times for a scene of moredate complexity and resolution. Once we consider the added cost of sampling for area light sources, depth of field, and photon mapping, the amount of computation required to render the scene can quickly become unmanageable.

Typically, the most expensive component of the raytracing process is the intersection testing between rays and objects. For a scene with N objects, the naive implementation of ray tracing we have been using thus far incurs a cost of $O(N)$ per ray for intersection testing. For complex scenes with polygonal meshes consisting of millions of triangles, this becomes the dominant factor determining rendering time.

Most of the time spent by the naive implementation is wasted. A ray will likely hit only a very small number of the objects in the scene. The important observation to make is that we should be able to quickly discard from consideration objects that have no chance whatsoever of being hit by the ray. If we do this cleverly, we can reduce by a large factor the number of intersection tests that need be carried out.

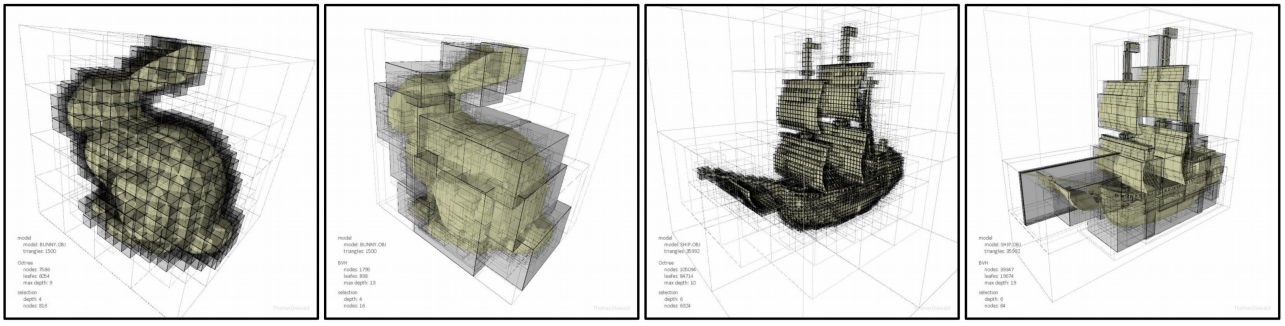
The most common acceleration techniques rely on space subdivision. The idea is to create a hierarchical structure that divides the scene into regions of progressively smaller volume (typically nested boxes of decreasing size), and to test a ray against these boxes to quickly determine what regions of the scene are traversed by the ray, so that only objects within those regions are actually checked for intersection.



Example of a space sub-division hierarchy in 2D (with bounding boxes). The scene S is progessively split into smaller regions forming a tree-shaped hierarchy. An incoming ray is tested against bounding boxes starting at S. In this example, the ray would be tested against $B1$, $B2$, then $B5$, and $B6$, and finally against $B7$ and $B8$. At $B7$ there is a list of objects the ray has to be tested against to determine intersection. For a complex scene, the hierarchical testing process quickly discards a large number of scene objects from having to be tested for intersection with the ray.

There are two commonly used methods for generating a spatial sub-division hierarchy: a) Octrees - which use cubic-shaped volumes to deivide the scene, and progressively divide each cube into 8 equal-sized cubic regions; and b) Bounded Volume Hierarchies (BVHs), which use appropriately sized bounding boxes to split the scene (similar to the example above). Each method has advantages and disadvantages. Octree partitioning is easy to produce since the subdivision process is

fixed and the bounding boxes for each subregion are easy to determine and unique. However, since octree boxes do not align with object bounding boxes, Octree hierarchies tend to require more subdivision to split objects in the scene from one another. BVHs on the other hand create boxes that fit object bounding boxes, and thus tend to require less splitting. However, each sub-region can be split in a number of different ways (in the example above, the scene could have been split so that B_1 and B_2 are chubby instead of tall, and this would produce a completely different hierarchy. There are specialized heuristics that can be used to guide the splitting process.



Corresponding Octree and BVH decompositions for two scenes. Note the regular lattice-like structure of the Octree hierarchy, and the visibly greated amount of boxes compared to BVH. Conversely, note the non-trivial arrangement of splits for the BVH hierarchy. Images courtesy of Thomas Diewald (`http://thomasdiewald.com/blog/?p=1488`)

While there is no general consensus about which of these subdivision methods performs best in a general scene, both provide very significant reduction of rendering time by reducing the number of intersection tests carried out for each ray. One final issue worth mentioning is that the amount of storage space required to maintain the Octree or BVH structure can grow very quickly, so in practice the speed-up obtained from a given amount of hierarchical splitting has to be balanced against the additional memory requirements.

```
Hierarchical space-subdivision procedure

Starting with the whole scene

  1.1) Given the current node and the list of objects contained in
       this node's volume.

  1.2) Compute a suitable split (either into octants for Octrees, or
       a pre-defined number of boxes for BVH).
```



```
  1.3) Insert each sub-region as a child of the current node

  1.4) Determine which objects in the current node are contained by
       each of the children, and update their corresponding list of
       objects removing them from the parent's list (to avoid
       unnecessary duplication).

  1.5) Recursively apply 1.1 to each of the children of this node
       until:

       1.5a) A pre-defined maximum depth has been reached
       1.5b) A sub-region contains no objects (remove it from the
             parent node's children list)
       1.5c) The number of objects contained in the subregion is
             at most a pre-defined maximum value (which could be 1)
```

```
Ray testing against a hierarchical space-subdivision structure

Starting at the root

  1.1) Test the ray for intersection against each the bounding
       box of each child of the current node

  1.2) If the ray intersects the boundig box

       1.2a) If the child is not a leaf node apply 1.1
             recursively to this child node

       1.2b) Test the ray for intersection against every object
             in the list of objects contained within this
             bounding box. Return the closest intersection
             (if any)

Note: The ray must be tested against every child of a node whose
      bounding box is intersected, as it is possible for a ray to
      cross multiple sub-regions for any given boundig box.
```

