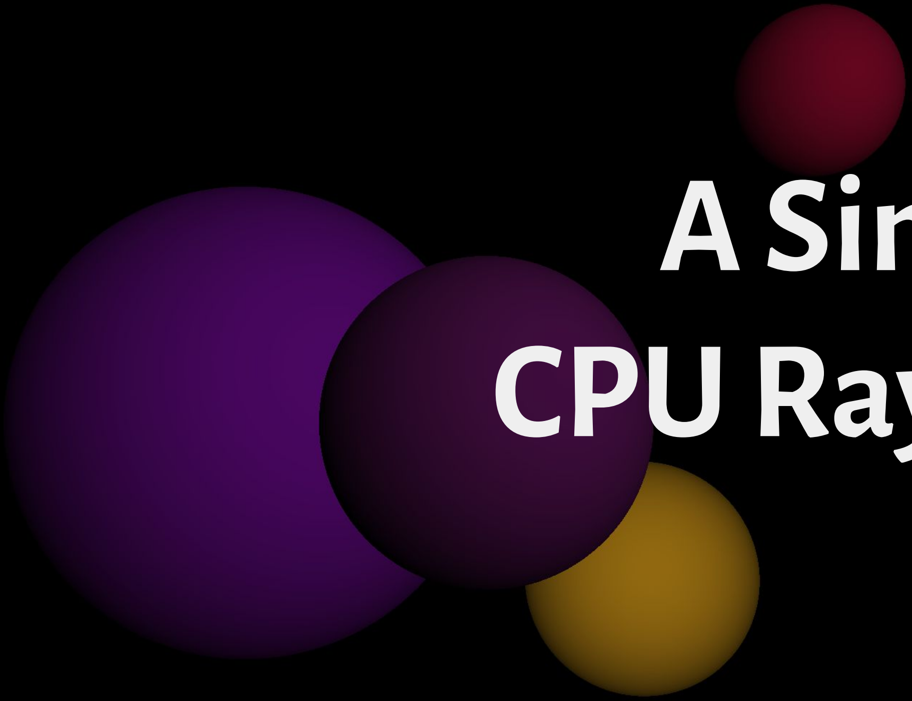
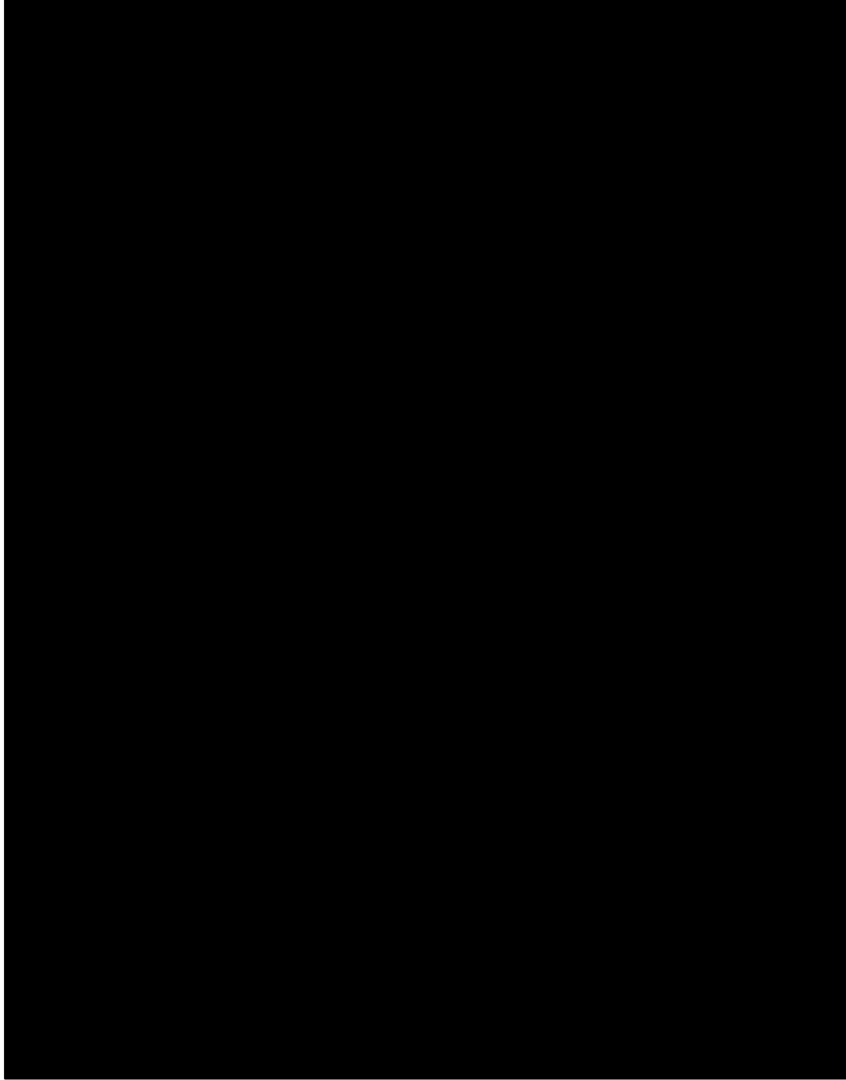


A Simple CPU Ray Tracer

Andrea Velasco

VU Effizientes Lösen von Partiellen Differentialgleich
University of Innsbruck

Index

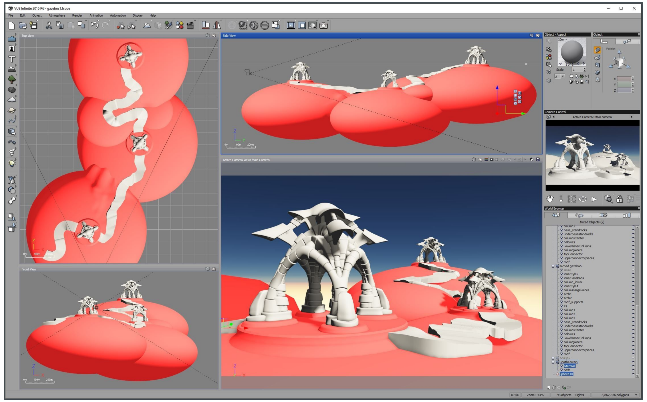


Rendering

The process by which a computer generates a 2D **image** of a **scene**.

Any image you see in a computer has been rendered in some way - even text.
The most common (and oldest) technique is called **rasterization**.

More advanced rendering techniques required for modern-day CGI, videogames, etc...

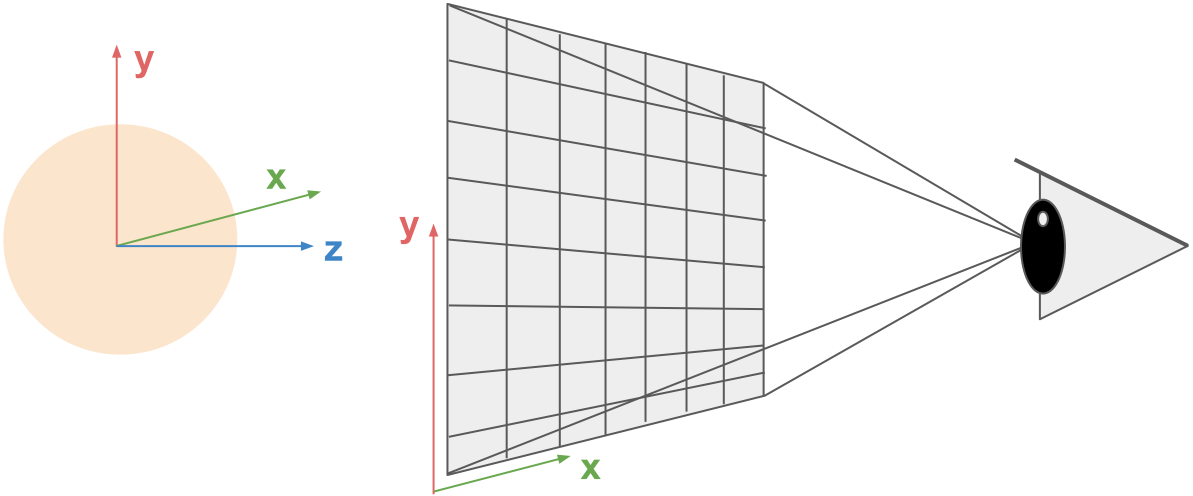


Ray Tracing

Simulation of light rays and their interactions with objects before they hit our eye.

Only difference is - we do it backwards.

Let’s define a 2D image and a viewer. We will call this structure a camera.

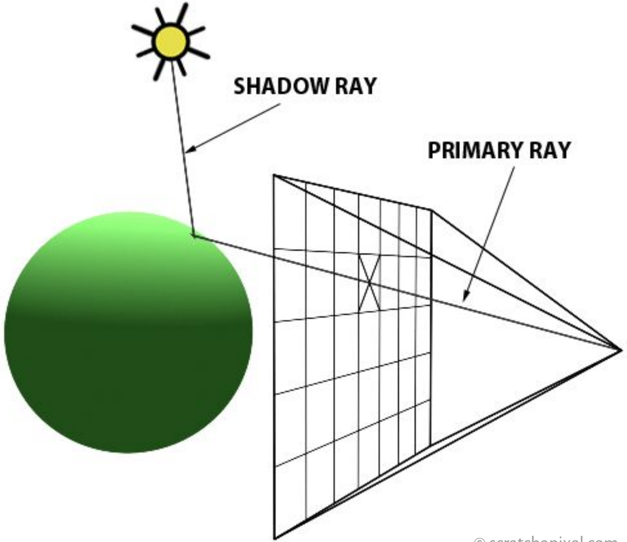


Ray Tracing

Since it would be ridiculous to simulate all the possible light rays that a source produces and see which ones reach the camera, let's instead go from camera to scene.

Algorithm:

```
for each pixel in image
{
   1. shoot ray from camera to center of pixel

   if( ray intersects object )
   {
     2. shoot ray from intersection point to light
     3. compute lighting
     4. image.at(pixel) = lighting * color
   }
 }
```

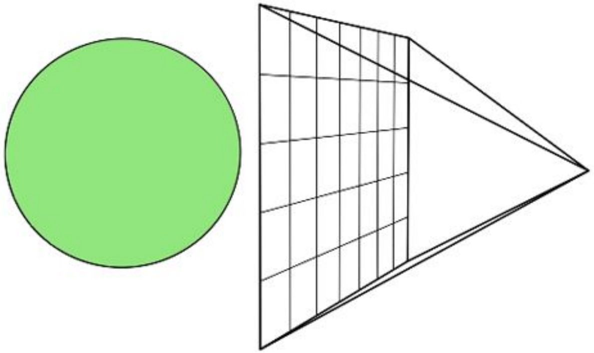


Ray Tracing

Since it would be ridiculous to simulate all the possible light rays that a source produces and see which ones reach the camera, let's instead go from camera to scene.

Algorithm:

```
for each pixel in image
{
   1. shoot ray from camera to center of pixel

   if( ray intersects object )
   {
     2. shoot ray from intersection point to light
     3. compute lighting
     4. image.at(pixel) = lighting * color
   }
 }
```



Implementation

Classes:

```
Camera
      Vec3d  _origin;
      Vec2i  _resolution;
      double _fov;

      map< vec<int>, Ray > getAllRays();
```

```
Ray
      Vec3d _origin;
      Vec3d _direction;
```

```
Sphere
      Vec3d  _center;
      double _radius;
      Vec3d  _color;

      bool intersects( const Ray &ray );
```

```
Light
      Vec3d _origin;
```

```
RayTracer
      Camera       _camera;
      vec<Sphere>  _spheres;
      Light        _light;
      ImageType    _image;

      void Update();
      ImageType getRender();
```

```cpp
void RayTracer::Update()
{
  cv::Mat depthBuffer;
  depthBuffer = cv::Mat::zeros(_camera.getResolution().x(), _camera.getResolution().y(), CV_64F) - 1e8;

  std::map<std::vector<int>, Ray> allRays = _camera.getAllRays();

  #pragma omp parallel for
  for( int i = 0; i < allRays.size(); i++ )
  {
    for( int s = 0; s < _spheres.size(); s++ )
    {
      Sphere currentSphere = _spheres[s];

      int x = i%_camera.getResolution().x();
      int y = (i - x) / _camera.getResolution().x();

      std::vector<int> key = {x,y};
      Ray &currentRay = allRays.at( key );

      Vector3d intersectionCoords;
      if( currentSphere.intersects(currentRay, intersectionCoords) )
      {
        // If this intersection point on this sphere is closest to camera so far, paint and update z buffer.
        if( intersectionCoords.z() > depthBuffer.at<double>(y, x) )
        {
          Vector3d surfaceNormal = (intersectionCoords - currentSphere.getCenter()).normalized();

          double   a = 2.0 * currentRay.getDirection().dot(surfaceNormal);
          Vector3d b = a * surfaceNormal;
          Vector3d finalDir = currentRay.getDirection() - b;

          Ray reflectedRay( intersectionCoords, finalDir );

          Vector3d intersectionToLight = (_light.getOrigin() - intersectionCoords).normalized();

          double ambient  = 0.5;
          double diffuse  = surfaceNormal.dot(intersectionToLight);
          double specular = 0.;

          double illum_total = (0.5 * ambient) + (0.6 * diffuse) + (0.2 * specular);

          _image.at<cv::Vec3b>(y, x) = illum_total * currentSphere.getColor();

          depthBuffer.at<double>(y, x) = intersectionCoords.z(); // update buffer with new closest z
        }
      }
    }
  }
}
```

Initialize depth buffer

```
void RayTracer::Update()
{
  cv::Mat depthBuffer;
  depthBuffer = cv::Mat::zeros(_camera.getResolution().x(), _camera.getResolution().y(), CV_64F) - 1e8;

  std::map<std::vector<int>, Ray> allRays = _camera.getAllRays();

  #pragma omp parallel for
  for( int i = 0; i < allRays.size(); i++ )
  {
    for( int s = 0; s < _spheres.size(); s++ )
    {
      Sphere currentSphere = _spheres[s];

      int x = i%_camera.getResolution().x();
      int y = (i - x) / _camera.getResolution().x();

      std::vector<int> key = {x,y};
      Ray &currentRay = allRays.at( key );

      Vector3d intersectionCoords;
      if( currentSphere.intersects(currentRay, intersectionCoords) )
      {
        // If this intersection point on this sphere is closest to camera so far, paint and update z buffer.
        if( intersectionCoords.z() > depthBuffer.at<double>(y, x) )
        {
          Vector3d surfaceNormal = (intersectionCoords - currentSphere.getCenter()).normalized();

          double   a = 2.0 * currentRay.getDirection().dot(surfaceNormal);
          Vector3d b = a * surfaceNormal;
          Vector3d finalDir = currentRay.getDirection() - b;

          Ray reflectedRay( intersectionCoords, finalDir );

          Vector3d intersectionToLight = (_light.getOrigin() - intersectionCoords).normalized();

          double ambient  = 0.5;
          double diffuse  = surfaceNormal.dot(intersectionToLight);
          double specular = 0.;

          double illum_total = (0.5 * ambient) + (0.6 * diffuse) + (0.2 * specular);

          _image.at<cv::Vec3b>(y, x) = illum_total * currentSphere.getColor();

          depthBuffer.at<double>(y, x) = intersectionCoords.z(); // update buffer with new closest z
        }
      }
    }
  }
}
```

Call OMP over a valid iteration

```cpp
void RayTracer::Update()
{
  cv::Mat depthBuffer;
  depthBuffer = cv::Mat::zeros(_camera.getResolution().x(), _camera.getResolution().y(), CV_64F) - 1e8;

  std::map<std::vector<int>, Ray> allRays = _camera.getAllRays();

  #pragma omp parallel for
  for( int i = 0; i < allRays.size(); i++ )
  {
    for( int s = 0; s < _spheres.size(); s++ )
    {
      Sphere currentSphere = _spheres[s];

      int x = i%_camera.getResolution().x();
      int y = (i - x) / _camera.getResolution().x();

      std::vector<int> key = {x,y};
      Ray &currentRay = allRays.at( key );

      Vector3d intersectionCoords;
      if( currentSphere.intersects(currentRay, intersectionCoords) )
      {
        // If this intersection point on this sphere is closest to camera so far, paint and update z buffer.
        if( intersectionCoords.z() > depthBuffer.at<double>(y, x) )
        {
          Vector3d surfaceNormal = (intersectionCoords - currentSphere.getCenter()).normalized();

          double   a = 2.0 * currentRay.getDirection().dot(surfaceNormal);
          Vector3d b = a * surfaceNormal;
          Vector3d finalDir = currentRay.getDirection() - b;

          Ray reflectedRay( intersectionCoords, finalDir );

          Vector3d intersectionToLight = (_light.getOrigin() - intersectionCoords).normalized();

          double ambient  = 0.5;
          double diffuse  = surfaceNormal.dot(intersectionToLight);
          double specular = 0.;

          double illum_total = (0.5 * ambient) + (0.6 * diffuse) + (0.2 * specular);

          _image.at<cv::Vec3b>(y, x) = illum_total * currentSphere.getColor();

          depthBuffer.at<double>(y, x) = intersectionCoords.z(); // update buffer with new closest z
        }
      }
    }
  }
}
```

Check for intersection,
return coordinates

```cpp
void RayTracer::Update()
{
  cv::Mat depthBuffer;
  depthBuffer = cv::Mat::zeros(_camera.getResolution().x(), _camera.getResolution().y(), CV_64F) - 1e8;

  std::map<std::vector<int>, Ray> allRays = _camera.getAllRays();

  #pragma omp parallel for
  for( int i = 0; i < allRays.size(); i++ )
  {
    for( int s = 0; s < _spheres.size(); s++ )
    {
      Sphere currentSphere = _spheres[s];

      int x = i%_camera.getResolution().x();
      int y = (i - x) / _camera.getResolution().x();

      std::vector<int> key = {x,y};
      Ray &currentRay = allRays.at( key );

      Vector3d intersectionCoords;
      if( currentSphere.intersects(currentRay, intersectionCoords) )
      {
        // If this intersection point on this sphere is closest to camera so far, paint and update z buffer.
        if( intersectionCoords.z() > depthBuffer.at<double>(y, x) )
        {
          Vector3d surfaceNormal = (intersectionCoords - currentSphere.getCenter()).normalized();

          double   a = 2.0 * currentRay.getDirection().dot(surfaceNormal);
          Vector3d b = a * surfaceNormal;
          Vector3d finalDir = currentRay.getDirection() - b;

          Ray reflectedRay( intersectionCoords, finalDir );

          Vector3d intersectionToLight = (_light.getOrigin() - intersectionCoords).normalized();

          double ambient  = 0.5;
          double diffuse  = surfaceNormal.dot(intersectionToLight);
          double specular = 0.;

          double illum_total = (0.5 * ambient) + (0.6 * diffuse) + (0.2 * specular);

          _image.at<cv::Vec3b>(y, x) = illum_total * currentSphere.getColor();

          depthBuffer.at<double>(y, x) = intersectionCoords.z(); // update buffer with new closest z
        }
      }
    }
  }
}
```

Check if intersection point is closer to camera than previous depth buffer value for this pixel

```cpp
void RayTracer::Update()
{
  cv::Mat depthBuffer;
  depthBuffer = cv::Mat::zeros(_camera.getResolution().x(), _camera.getResolution().y(), CV_64F) - 1e8;

  std::map<std::vector<int>, Ray> allRays = _camera.getAllRays();

  #pragma omp parallel for
  for( int i = 0; i < allRays.size(); i++ )
  {
    for( int s = 0; s < _spheres.size(); s++ )
    {
      Sphere currentSphere = _spheres[s];

      int x = i%_camera.getResolution().x();
      int y = (i - x) / _camera.getResolution().x();

      std::vector<int> key = {x,y};
      Ray &currentRay = allRays.at( key );

      Vector3d intersectionCoords;
      if( currentSphere.intersects(currentRay, intersectionCoords) )
      {
        // If this intersection point on this sphere is closest to camera so far, paint and update z buffer.
        if( intersectionCoords.z() > depthBuffer.at<double>(y, x) )
        {
          Vector3d surfaceNormal = (intersectionCoords - currentSphere.getCenter()).normalized();

          double   a = 2.0 * currentRay.getDirection().dot(surfaceNormal);
          Vector3d b = a * surfaceNormal;
          Vector3d finalDir = currentRay.getDirection() - b;

          Ray reflectedRay( intersectionCoords, finalDir );

          Vector3d intersectionToLight = (_light.getOrigin() - intersectionCoords).normalized();

          double ambient  = 0.5;
          double diffuse  = surfaceNormal.dot(intersectionToLight);
          double specular = 0.;

          double illum_total = (0.5 * ambient) + (0.6 * diffuse) + (0.2 * specular);

          _image.at<cv::Vec3b>(y, x) = illum_total * currentSphere.getColor();

          depthBuffer.at<double>(y, x) = intersectionCoords.z(); // update buffer with new closest z
        }
      }
    }
  }
}
```

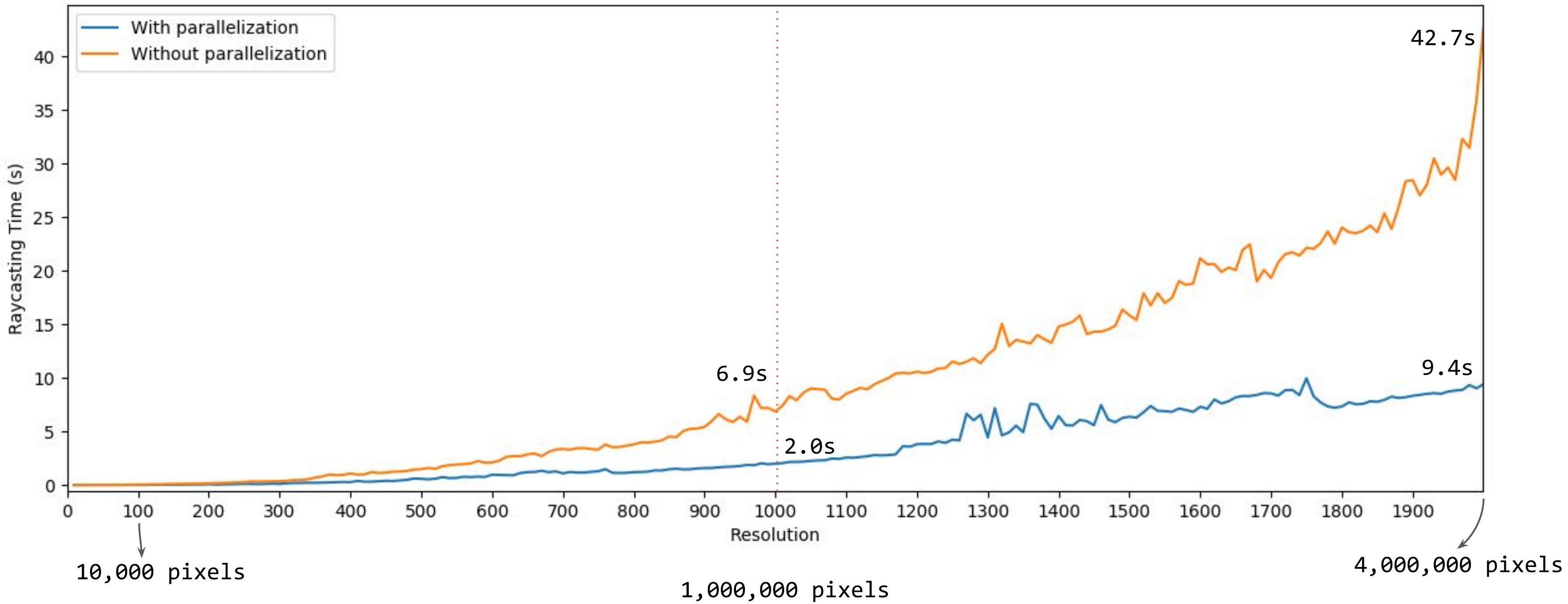
Calculate illumination

```
void RayTracer::Update()
{
  cv::Mat depthBuffer;
  depthBuffer = cv::Mat::zeros(_camera.getResolution().x(), _camera.getResolution().y(), CV_64F) - 1e8;

  std::map<std::vector<int>, Ray> allRays = _camera.getAllRays();

  #pragma omp parallel for
  for( int i = 0; i < allRays.size(); i++ )
  {
    for( int s = 0; s < _spheres.size(); s++ )
    {
      Sphere currentSphere = _spheres[s];

      int x = i%_camera.getResolution().x();
      int y = (i - x) / _camera.getResolution().x();

      std::vector<int> key = {x,y};
      Ray &currentRay = allRays.at( key );

      Vector3d intersectionCoords;
      if( currentSphere.intersects(currentRay, intersectionCoords) )
      {
        // If this intersection point on this sphere is closest to camera so far, paint and update z buffer.
        if( intersectionCoords.z() > depthBuffer.at<double>(y, x) )
        {
          Vector3d surfaceNormal = (intersectionCoords - currentSphere.getCenter()).normalized();

          double   a = 2.0 * currentRay.getDirection().dot(surfaceNormal);
          Vector3d b = a * surfaceNormal;
          Vector3d finalDir = currentRay.getDirection() - b;

          Ray reflectedRay( intersectionCoords, finalDir );

          Vector3d intersectionToLight = (_light.getOrigin() - intersectionCoords).normalized();

          double ambient  = 0.5;
          double diffuse  = surfaceNormal.dot(intersectionToLight);
          double specular = 0.;

          double illum_total = (0.5 * ambient) + (0.6 * diffuse) + (0.2 * specular);

          _image.at<cv::Vec3b>(y, x) = illum_total * currentSphere.getColor();

          depthBuffer.at<double>(y, x) = intersectionCoords.z(); // update buffer with new closest z
        }
      }
    }
  }
}
```

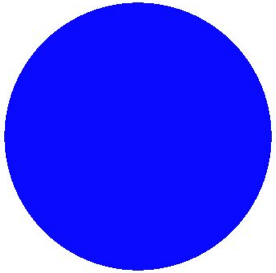
Update image and depth buffer

Performance Comparison

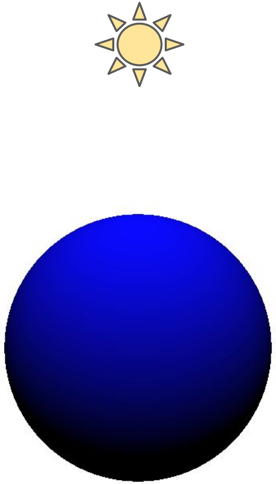
Starts to become very noticeable at 1000 x 1000, and the times continue to diverge as the resolution increases.



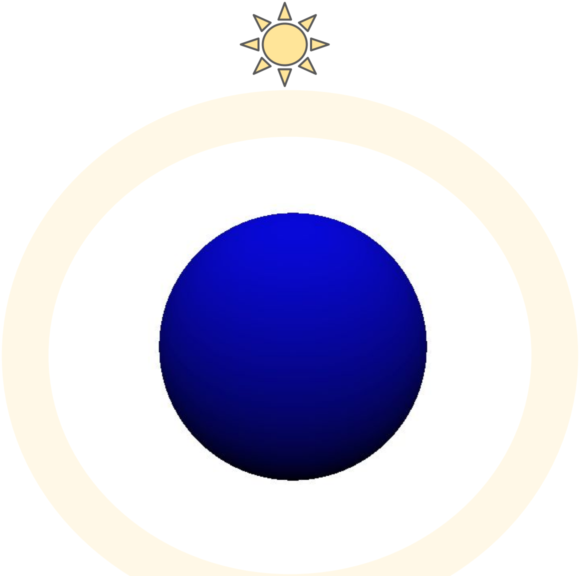
Lighting & Results



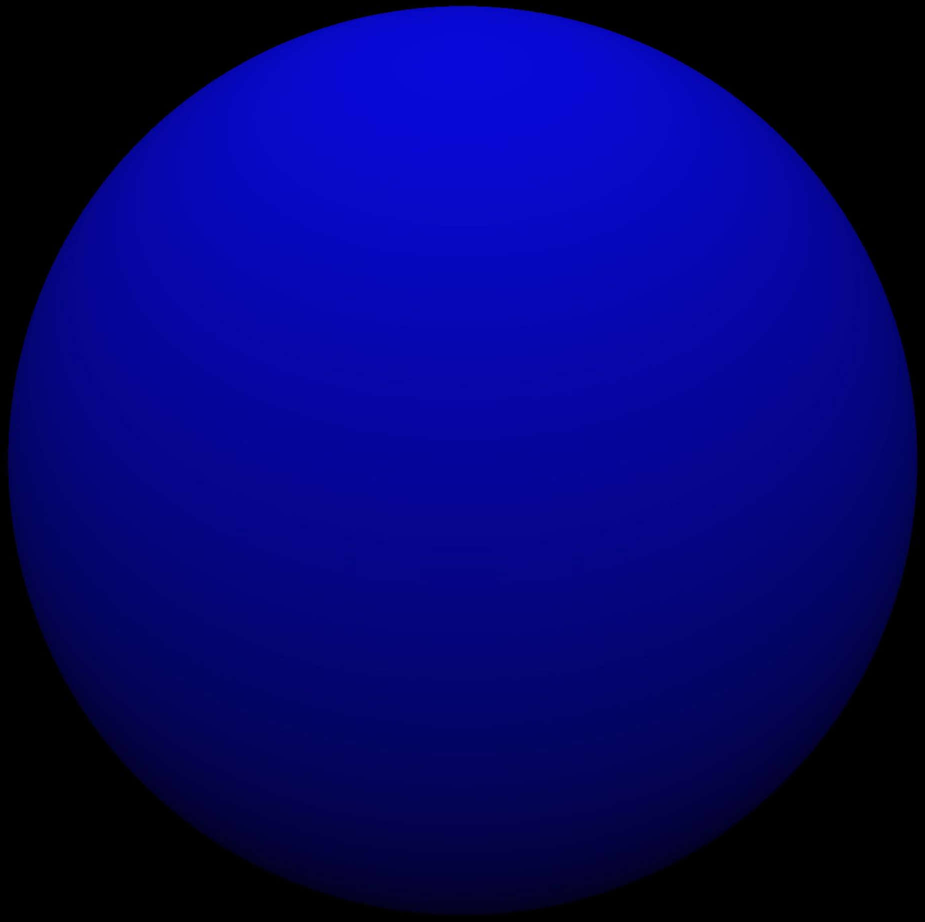
No lighting
Just paint pixel if there's an intersection.

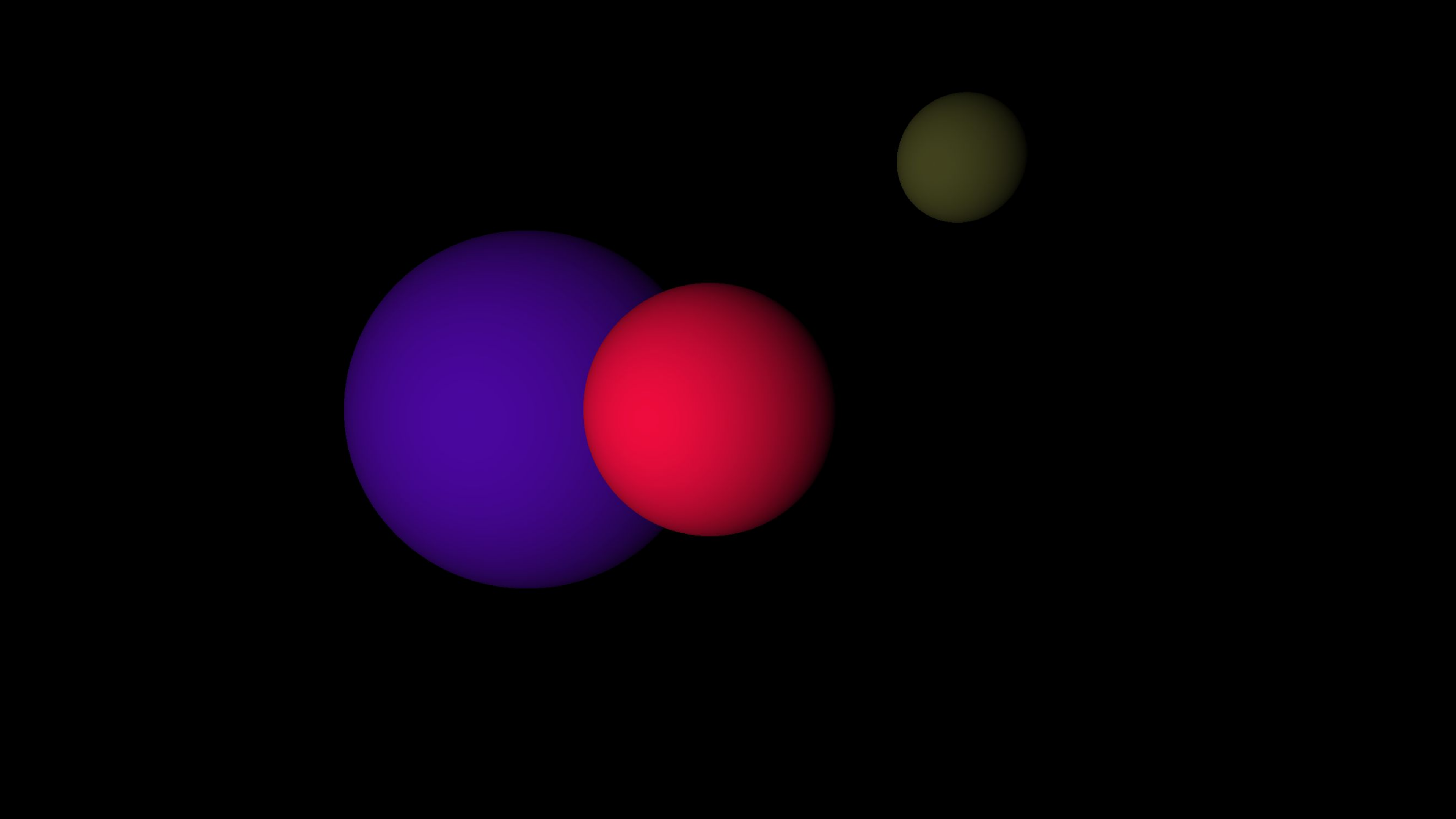


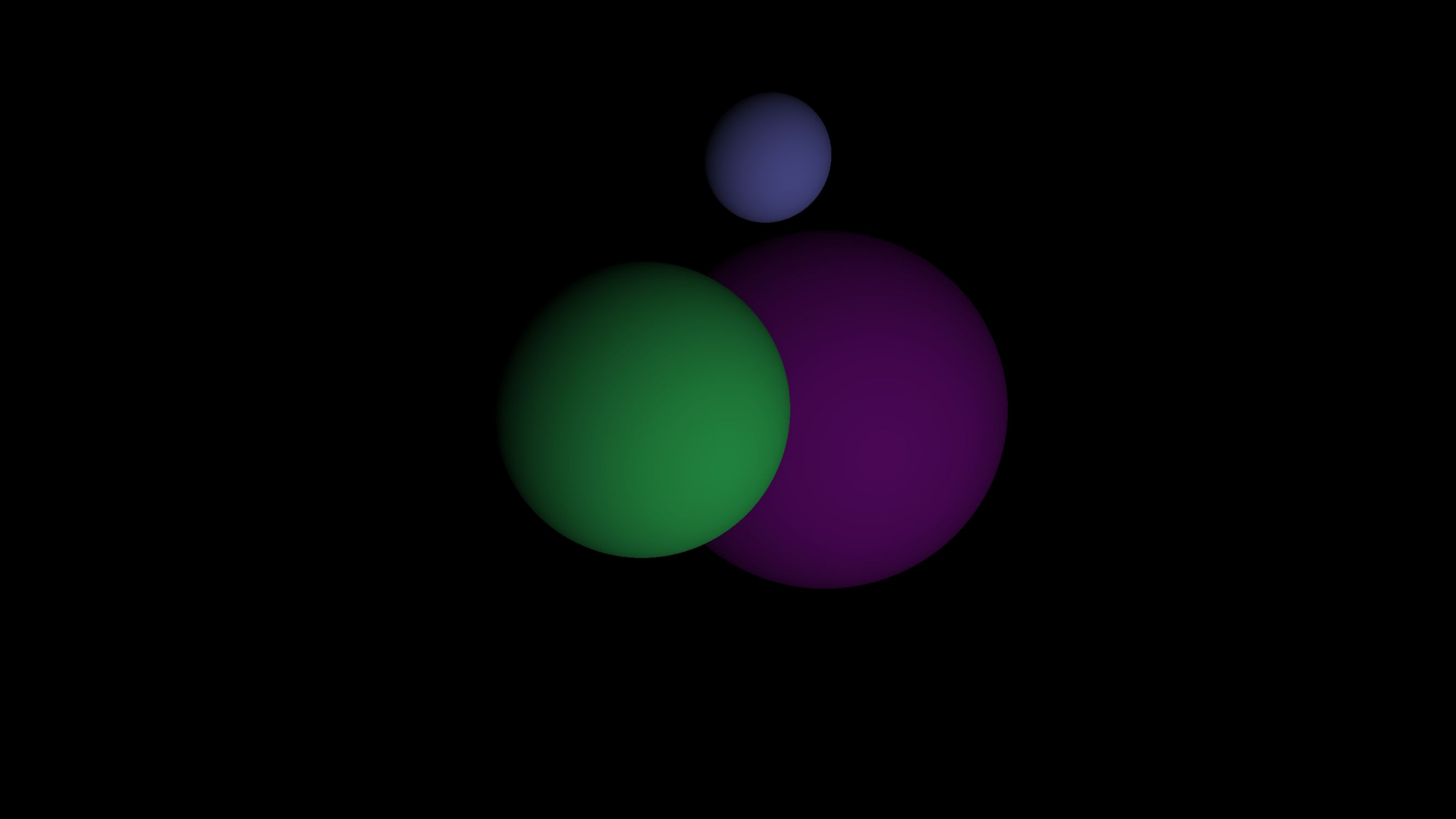
Diffuse Lighting



Diffuse + **Ambient** Lighting







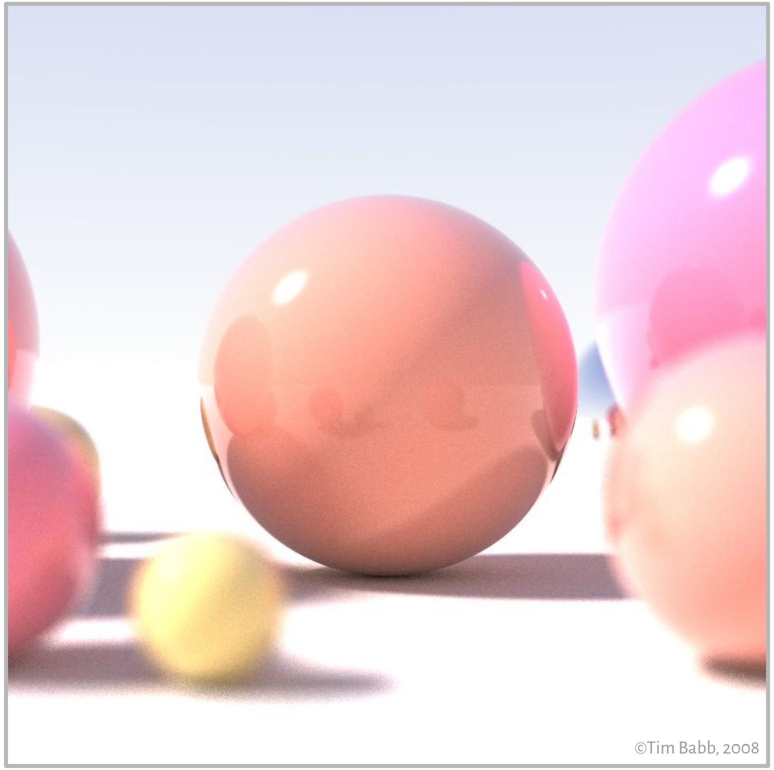
Improvements

Much to be done still with primitives:

- Shadows
- Reflection/refraction, specularity
- Image Based Lighting
- Depth of Field

Then one can move on to...

- Meshes
  - Triangle-Ray intersection
  - UV coordinates



State of The Art

"Ray tracing isn't too slow; computers are too slow." (Kajiya 1986)

- Becoming less and less true nowadays
  - Have much more computational power at hardware level (CPUs & GPUs)
  - New techniques to reduce amount of work needed
    - Hierarchy of bounding volumes to partition space to reduce amount of ray-primitive intersection tests
    - Divide-and-Conquer Ray Tracing Algorithm (Mora, 2011)
      - Subdivides packets of rays & objects together for intense parallelization
  - Real-time ray tracing is a really popular topic
    - NVIDIA dedicated hardware
    - Unity, Unreal Engine integrations

Image rendered by Gilles Tran