1.

1.1

```
IndexType num_nodes = span_right - span_left;
if (num_nodes <= 4 || depth >= CUTOFF_DEPTH)
{
  InternalNode result(span_left, span_right);
  result.is_leaf = true;
  result.aabb    = prebuilt_aabb;
  internal_nodes.push_back(result);
  return internal_nodes.size() - 1;
}
```

This code implements the stop criteria for BVH construction. When the number of primitives in the current node is ≤ 4 or the recursion depth exceeds `CUTOFF_DEPTH`, the node is marked as a leaf node. The computed bounding box is stored for pruning during traversal.

1.2

```
std::nth_element(
    nodes.begin() + span_left,
    nodes.begin() + split,
    nodes.begin() + span_right,
    [&](const NodeType& a, const NodeType& b) {
        AABB a_aabb = a.getAABB();
        AABB b_aabb = b.getAABB();
        Float a_centroid_dim = a_aabb.low_bnd[dim] + a_aabb.upper_bnd[dim];
        Float b_centroid_dim = b_aabb.low_bnd[dim] + b_aabb.upper_bnd[dim];
        return a_centroid_dim < b_centroid_dim;
    }
);
```

This implements the Median Heuristic for BVH splitting. Objects are partitioned according to the centroid along the longest axis using `std::nth_element`

2.

```
Float t_in, t_out;
if (!node.aabb.intersect(ray, &t_in, &t_out)) return result;

if (node.is_leaf) {
  for (IndexType span_index = node.span_left; span_index < node.span_right;
++span_index){
    result |= callback(ray, nodes[span_index].getData());
  }
}
```

Ray-AABB intersection is used to prune irrelevant BVH nodes. If the ray does not intersect the bounding box, traversal stops early.

3.

```
for (int sample = 0; sample < spp; sample++) {
    const Vec2f &pixel_sample = sampler.getPixelSample();
    auto ray = camera->generateDifferentialRay(pixel_sample.x, pixel_sample.y);
    const Vec3f &L = Li(scene, ray, sampler);
    camera->getFilm()->commitSample(pixel_sample, L);
}
```

Each pixel is sampled `spp` times with random sub-pixel offsets via `getPixelSample()`. The radiance is accumulated for Monte Carlo integration.

4.

```
if (scene->intersect(shadow_ray, shadow_interaction)) {
    return Vec3f(0.0f);
}
```

```
const Vec3f fr = interaction.bsdf->evaluate(interaction);
const Float cos_theta_at_surface = std::abs(Dot(interaction.normal,
to_light_dir));
const Float cos_theta_at_light = std::abs(Dot(light_normal, -to_light_dir));
const Float G = cos_theta_at_surface * cos_theta_at_light / dist_sq;
return Le * fr * G / pdf;
```

```
if (bsdf != nullptr && is_ideal_diffuse) {
  // if (false) {
    // TODO(HW3): Compute the contribution
    //
    // You can use bsdf->evaluate(interaction) * cos_theta to approximate the
    // albedo. In this homework, we do not need to consider a
    // radiometry-accurate model, so a simple phong-shading-like model is can be
    // used to determine the value of color.

    // The angle between light direction and surface normal
    Float cos_theta =
        std::max(Dot(light_dir, interaction.normal), 0.0f);  // one-sided
    // Float cos_theta = std::abs(Dot(light_dir, interaction.normal));

    // You should assign the value to color
    // color = ...
    // color = bsdf->evaluate(interaction) * cos_theta;
    auto albedo = bsdf->evaluate(interaction) * cos_theta;
    accumulated_color += albedo * point_light_flux  / (4 * PI * dist_to_light *
dist_to_light);
  }

  const Vec3f point_light_position_2 = {0.0f, 0.0f, 5.0f};
```

```
  const Float dist_to_light_2 = Norm(point_light_position_2 - interaction.p);
  const Vec3f light_dir_2     = Normalize(point_light_position_2 -
interaction.p);

  auto shadow_ray_2 = DifferentialRay(interaction.p, light_dir_2,
RAY_DEFAULT_MIN, dist_to_light_2 - RAY_DEFAULT_MIN);

  SurfaceInteraction shadow_interaction_2;

  if (!scene->intersect(shadow_ray_2, shadow_interaction_2)) {
      const BSDF *bsdf = interaction.bsdf;
      if (dynamic_cast<const IdealDiffusion *>(bsdf) != nullptr) {
          const Float cos_theta_2 = std::max(Dot(light_dir_2,
interaction.normal), 0.0f);
          const auto albedo = bsdf->evaluate(interaction) * cos_theta_2;
          accumulated_color += albedo * point_light_flux / (4 * PI *
dist_to_light_2 * dist_to_light_2);
      }
  }

  return accumulated_color;
```

A ray is shot from the surface point toward a sampled point on the light source. If this ray hits any object before reaching the light, the light is considered blocked and contributes zero intensity. And the surface color (albedo) is divided by π to produce a physically correct diffuse reflectance.

5.

```
bool tir = !Refract(interaction.wo, normal, eta_corrected, interaction.wi);
if (tir) {
  interaction.wi = Reflect(interaction.wo, normal);
}
if (pdf != nullptr) *pdf = 1.0F;
return Vec3f(1.0);
```

This implements physical refraction using Snell's Law. If Total Internal Reflection (TIR) occurs, reflection is used instead.

6.

```
if (is_perfect_refraction) {
    Float pdf;
    interaction.bsdf->sample(interaction, sampler, &pdf);
    ray = interaction.spawnRay(interaction.wi);
    continue;
}
```

The integrator detects refractive surfaces via RTTI and continues ray tracing with the updated refracted direction, ensuring correct light transport behavior.

7.

```
const Vec3f point_light_position_2 = {0.0f, 0.0f, 5.0f};

  const Float dist_to_light_2 = Norm(point_light_position_2 - interaction.p);
  const Vec3f light_dir_2     = Normalize(point_light_position_2 -
interaction.p);

  auto shadow_ray_2 = DifferentialRay(interaction.p, light_dir_2,
RAY_DEFAULT_MIN, dist_to_light_2 - RAY_DEFAULT_MIN);

  SurfaceInteraction shadow_interaction_2;

  if (!scene->intersect(shadow_ray_2, shadow_interaction_2)) {
      const BSDF *bsdf = interaction.bsdf;
      if (dynamic_cast<const IdealDiffusion *>(bsdf) != nullptr) {
          const Float cos_theta_2 = std::max(Dot(light_dir_2,
interaction.normal), 0.0f);
          const auto albedo = bsdf->evaluate(interaction) * cos_theta_2;
          accumulated_color += albedo * point_light_flux / (4 * PI *
dist_to_light_2 * dist_to_light_2);
      }
  }
```

This implements multiple lights.

8.

```
Vec3f IntersectionTestIntegrator::directLighting(
    ref<Scene> scene, SurfaceInteraction &interaction, Sampler &sampler) const {

    const Vec3f light_origin = {-0.25f, 1.98f, -0.25f};
    const Vec3f light_edge1 = {0.5f, 0.0f, 0.0f};
    const Vec3f light_edge2 = {0.0f, 0.0f, 0.5f};
    const Vec3f light_radiance = {17.0f, 12.0f, 4.0f};

    const Vec3f light_normal = Normalize(Cross(light_edge1, light_edge2));
    const Float light_area = Norm(Cross(light_edge1, light_edge2));
    const Float light_pdf = 1.0f / light_area;

    const Vec2f &rand_sample = sampler.get2D();
    const Vec3f point_on_light = light_origin + rand_sample.x * light_edge1 +
rand_sample.y * light_edge2;

    const Vec3f to_light_dir = Normalize(point_on_light - interaction.p);
    const Float dist_sq = Dot(point_on_light - interaction.p, point_on_light -
interaction.p);
    const Float dist = std::sqrt(dist_sq);

    auto shadow_ray = DifferentialRay(interaction.p, to_light_dir,
RAY_DEFAULT_MIN, dist - RAY_DEFAULT_MIN);
    SurfaceInteraction shadow_interaction;
    if (scene->intersect(shadow_ray, shadow_interaction)) {
        return Vec3f(0.0f);
    }
```

```
    const Vec3f Le = light_radiance;

    interaction.wi = to_light_dir;
    const Vec3f fr = interaction.bsdf->evaluate(interaction);
    const Float cos_theta_at_surface = std::abs(Dot(interaction.normal,
to_light_dir));
    const Float cos_theta_at_light = std::abs(Dot(light_normal, -to_light_dir));
    const Float G = cos_theta_at_surface * cos_theta_at_light / dist_sq;

    const Float pdf = light_pdf;

    return Le * fr * G / pdf;
}
```

This implement rectangle light.