Raytracing: la percezione creativa della luce

Lorenzo Piarulli 1884766

Le scene ed i modelli aggiuntivi utilizzati sono stati presi dal gioco mobile "Gravio" ideato e realizzato dal sottoscritto.

Link del gioco sulla piattaforma Play Store

Experiments

Facile

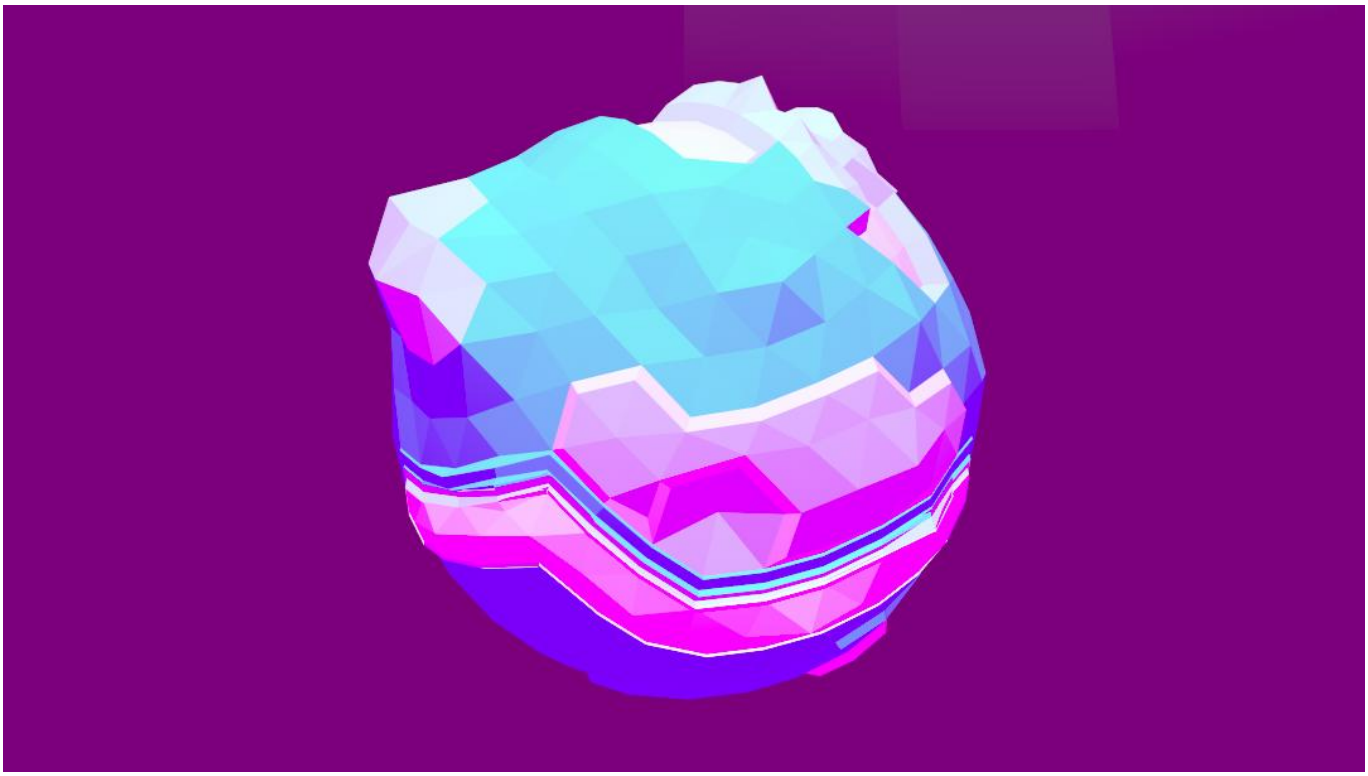
Nel primo paragrafo viene descritto quello che è stato il primo approccio al raytracing: sperimentare. Nelle prossime linee di codice vengono descritti algoritmi che mescolano e giocano con gli elementi teorici studiati.

Negative colors and dedicated color shadow

L'algoritmo presentato restituisce una scena i cui i colori assegnati vengono invertiti e l'ombreggiatura viene applicata alla sola coordinata y, che restituisce una tonalità violacea. Il prodotto scalare calcolato per rappresentare le ombre può essere applicato ad ogni altra coordinata, questa variazione restituirà scene con ombre di tonalità differenti.

```
static vec4f shade_exp1(const scene_data& scene, const bvh_scene& bvh,
    const ray3f& ray, int bounce, rng_state& rng,
    const raytrace_params& params) {
  auto isec = intersect_bvh(bvh, scene, ray);
  if (!isec.hit) {
    return {0, 0, 0, 0};
  }

  const auto& instance = scene.instances[isec.instance];
  const auto& shape    = scene.shapes[instance.shape];
  const auto& material = scene.materials[instance.material];
  const auto& normal   = eval_normal(shape, isec.element, isec.uv);

  auto position = transform_point(
      instance.frame, eval_position(shape, isec.element, isec.uv));
  vec4f color = rgb_to_rgba(material.color);
  vec4f nor   = {normal.x, normal.y, normal.z, 0};
  vec4f ray4d = {ray.d.x, ray.d.y, ray.d.z, 0};

  auto mod = dot(normal, -ray.d);

  return {(1 - color.x), (1 - color.y) * mod, (1 - color.z), color.w};
}
```

Di seguito i risultati ottenuti



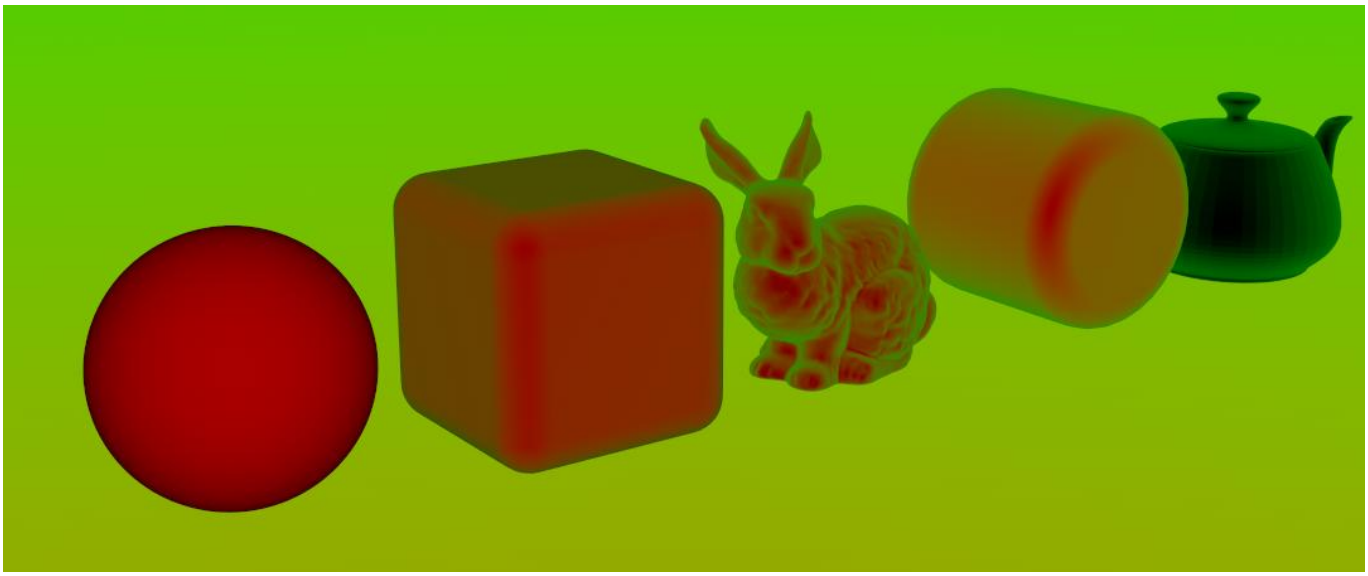
More than one type of shadow

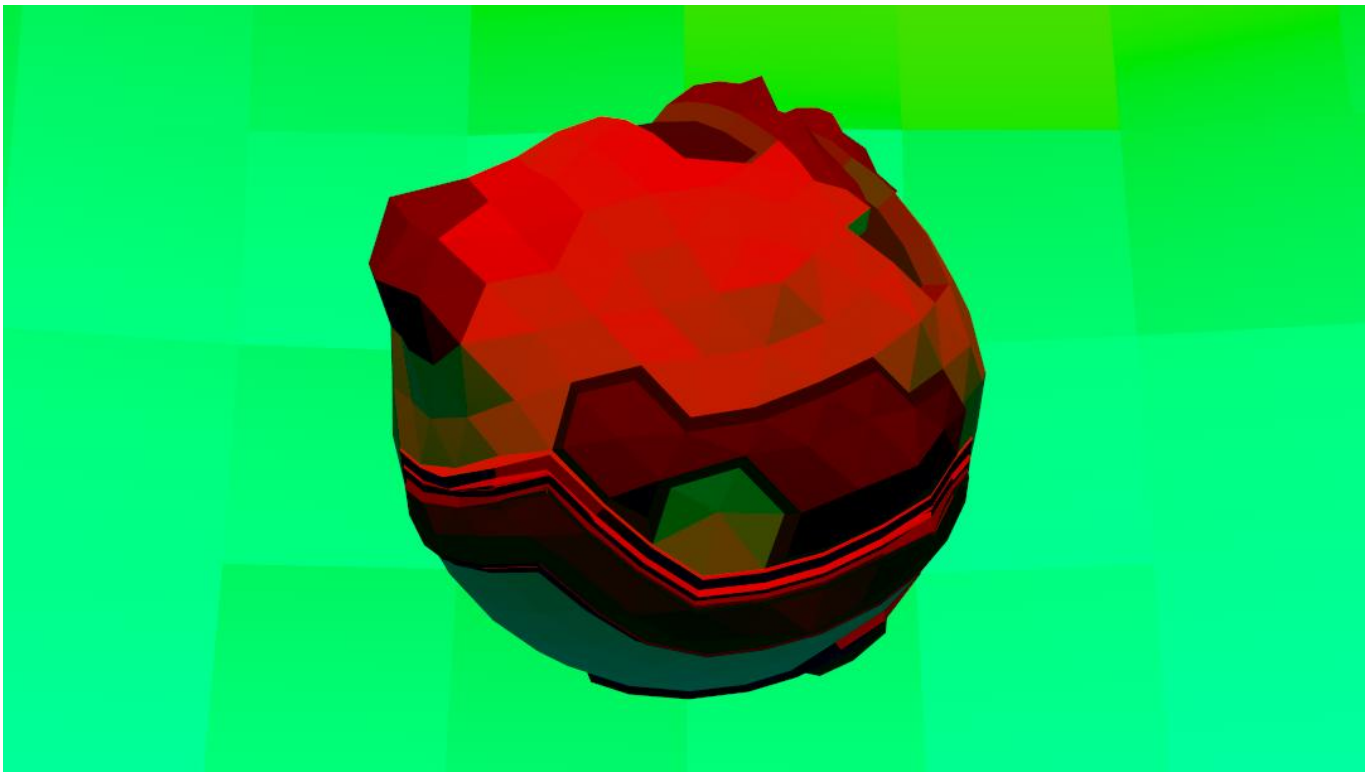
Il secondo algoritmo che viene presentato gioca sull'utilizzo del prodotto scalare, infatti moltiplica ad ogni coordinata del colore un differente tipo di rappresentazione delle ombre. Nel primo caso si ha una rappresentazione naturale, nel secondo viene presentata invertita e nell'ultimo il prodotto viene calcolato tra il vettore normale ed il vettore direzione.

```
static vec4f shade_exp2(const scene_data& scene, const bvh_scene& bvh,
    const ray3f& ray, int bounce, rng_state& rng,
    const raytrace_params& params) {
  auto isec = intersect_bvh(bvh, scene, ray);
  if (!isec.hit) {
    return {0, 0, 0, 0};
  }
```

```
    const auto& instance = scene.instances[isec.instance];
    const auto& shape    = scene.shapes[instance.shape];
    const auto& material = scene.materials[instance.material];
    const auto& normal   = eval_normal(shape, isec.element, isec.uv);

    auto position = transform_point(
        instance.frame, eval_position(shape, isec.element, isec.uv));
    vec4f color = rgb_to_rgba(material.color);
    vec4f nor   = {normal.x, normal.y, normal.z, 0};
    vec4f ray4d = {ray.d.x, ray.d.y, ray.d.z, 0};

    auto mod  = dot(normal, -ray.d);
    auto mod1 = 1 - dot(normal, -ray.d);
    auto mod2 = dot(ray.d, normal);

    return {color.x * mod, color.y * mod1, color.z * mod2};
  }
```

Le immagini ottenute sono le seguenti:





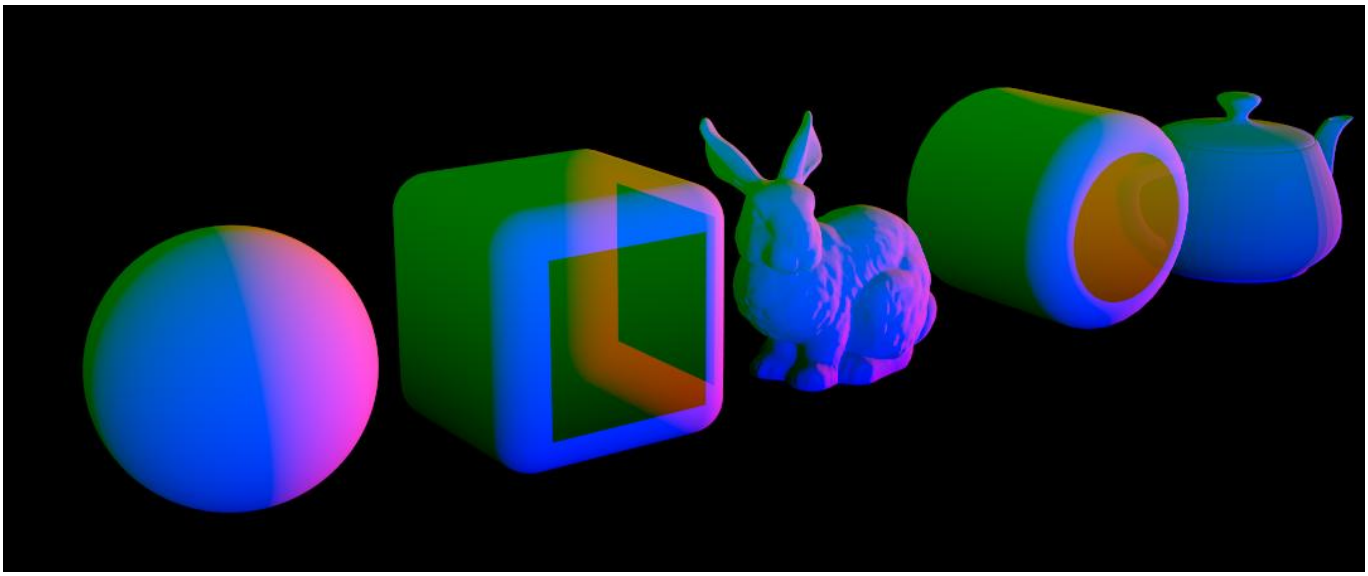
Recursive representation of normal and position vectors

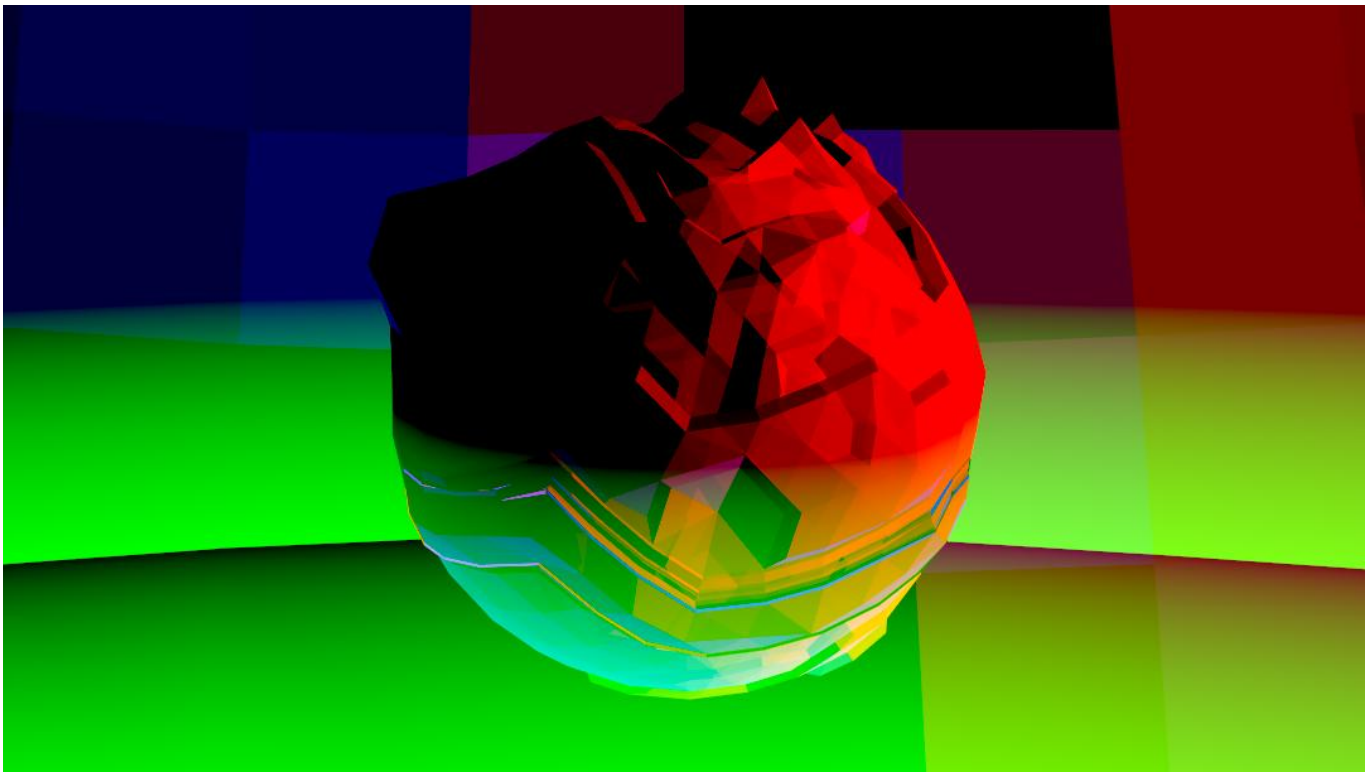
L'algoritmo realizzato di seguito, utilizza un'idea differente dagli altri: applica una struttura ricorsiva. La ricorsione permette ai raggi di rimbalzare e registrare sulla superficie (dove è avvenuto il rimbalzo) i valori del prossimo materiale che incontrerà. In questo caso il raggio, dopo essere rimbalzato, continua per la sua direzione creando un'effetto di trasparenza. Inoltre, i valori che vengono restituiti nella prima intersezione non sono i colori bensì i vettori normale e posizione.

```
static vec4f shade_exp3(const scene_data& scene, const bvh_scene& bvh,
    const ray3f& ray, int bounce, rng_state& rng,
    const raytrace_params& params) {
  auto isec = intersect_bvh(bvh, scene, ray);
  if (!isec.hit) {
    return {0, 0, 0, 0};
```

```
  }

  const auto& instance = scene.instances[isec.instance];
  const auto& shape    = scene.shapes[instance.shape];
  const auto& material = scene.materials[instance.material];
  const auto& normal   = eval_normal(shape, isec.element, isec.uv);

  auto position = transform_point(
      instance.frame, eval_position(shape, isec.element, isec.uv));
  vec4f color = rgb_to_rgba(material.color);
  vec4f nor   = {normal.x, normal.y, normal.z, 0};
  vec4f ray4d = {ray.d.x, ray.d.y, ray.d.z, 0};

  auto modx = fmod(normal.x, 1);
  auto mody = fmod(position.y, 1);
  auto modz = fmod(normal.z, 1);

  auto ret = vec4f{modx, mody, modz, 0};

  if (bounce >= params.bounces) {
    return ret;
  }

  ret += color * shade_exp3(scene, bvh, ray3f{position, ray.d}, bounce + 1,
rng,
                    params);
  return ret;
}
```

Le scene ottenute sono le seguenti:





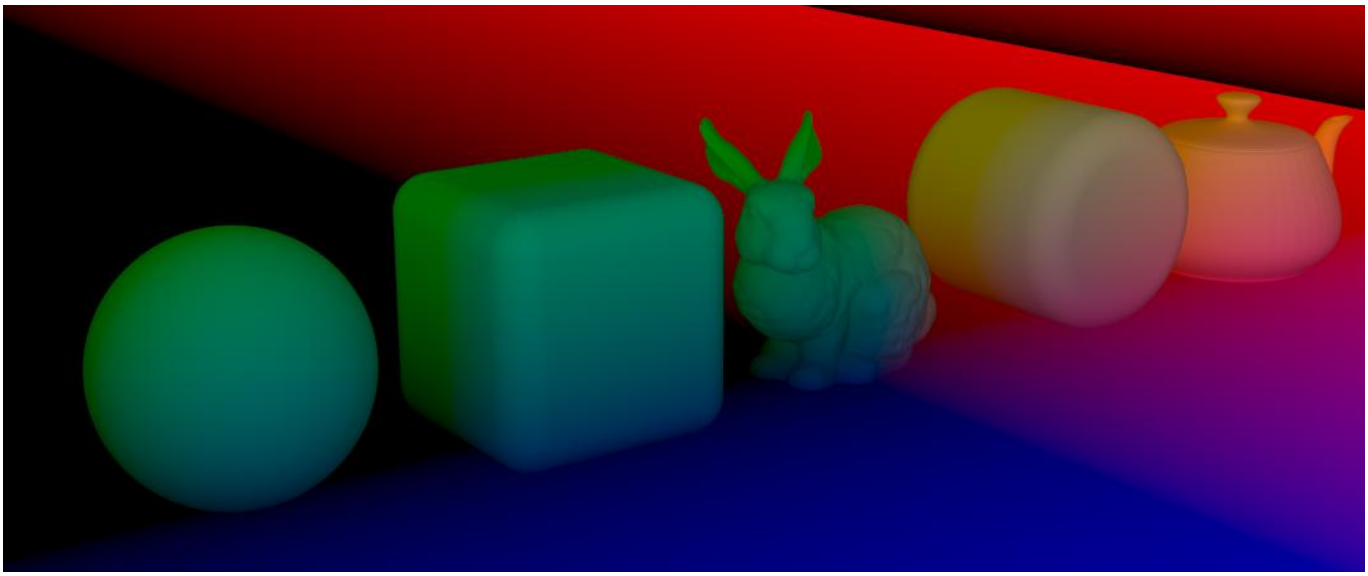
Normal vector representation vs Position vector representation

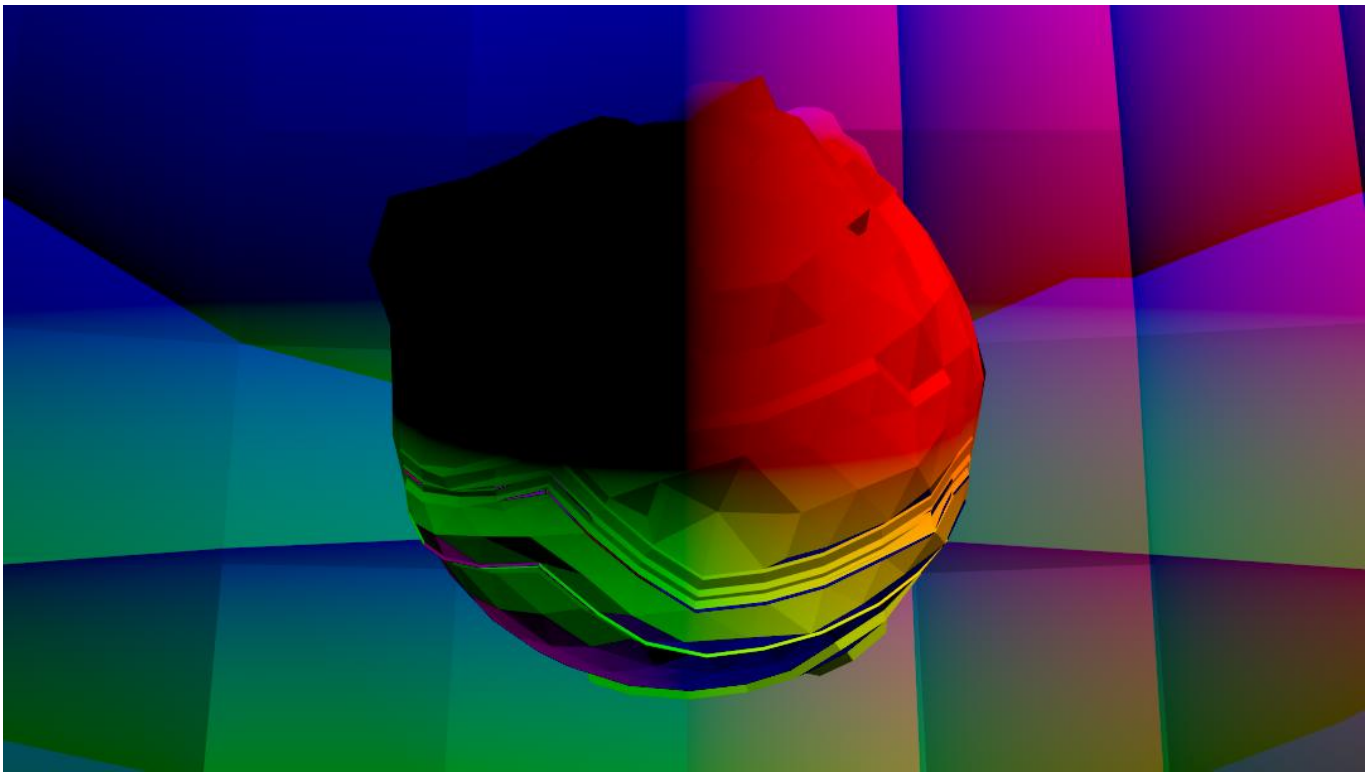
In quest'ultimo algoritmo, vengono restituiti due vettori fondamentali: il vettore posizione ed il vettore normale. I valori ottenuti, utilizzando il vettore posizione, vengono moltiplicati per un valore *smooth* che restituisce maggiore corpo agli oggetti rappresentati, tale valore viene aumentato di 0.5 per incrementare la luminosità.

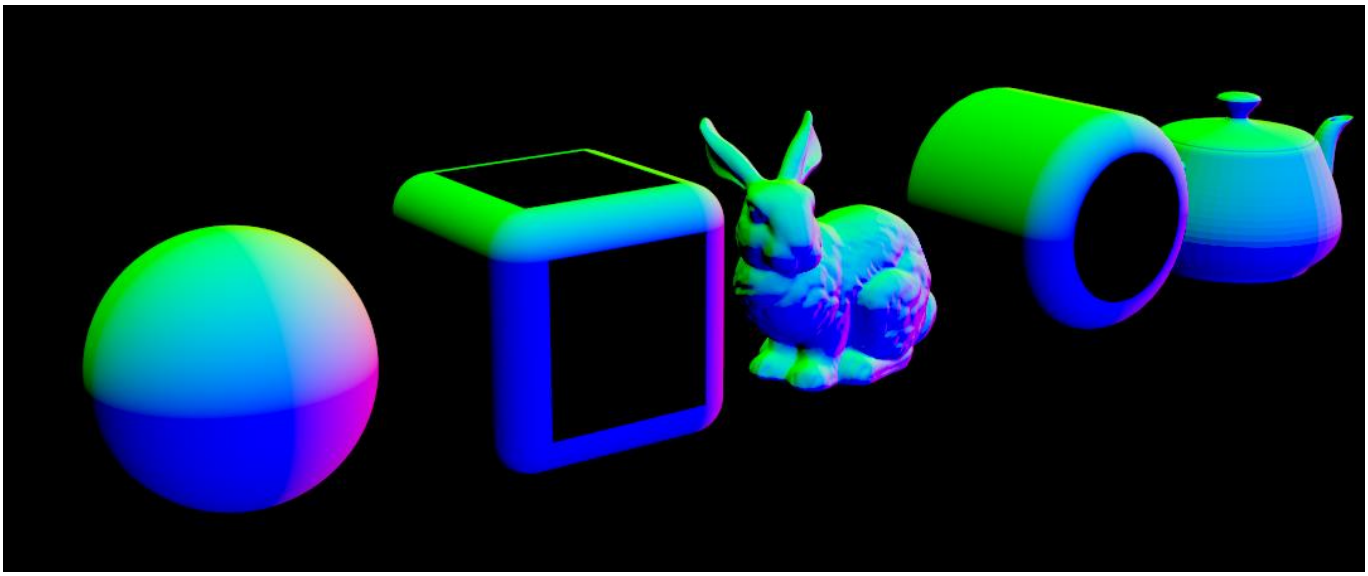
```
static vec4f shade_exp4(const scene_data& scene, const bvh_scene& bvh,
    const ray3f& ray, int bounce, rng_state& rng,
    const raytrace_params& params) {
  auto isec = intersect_bvh(bvh, scene, ray);
  if (!isec.hit) {
    return {0, 0, 0, 0};
  }
```

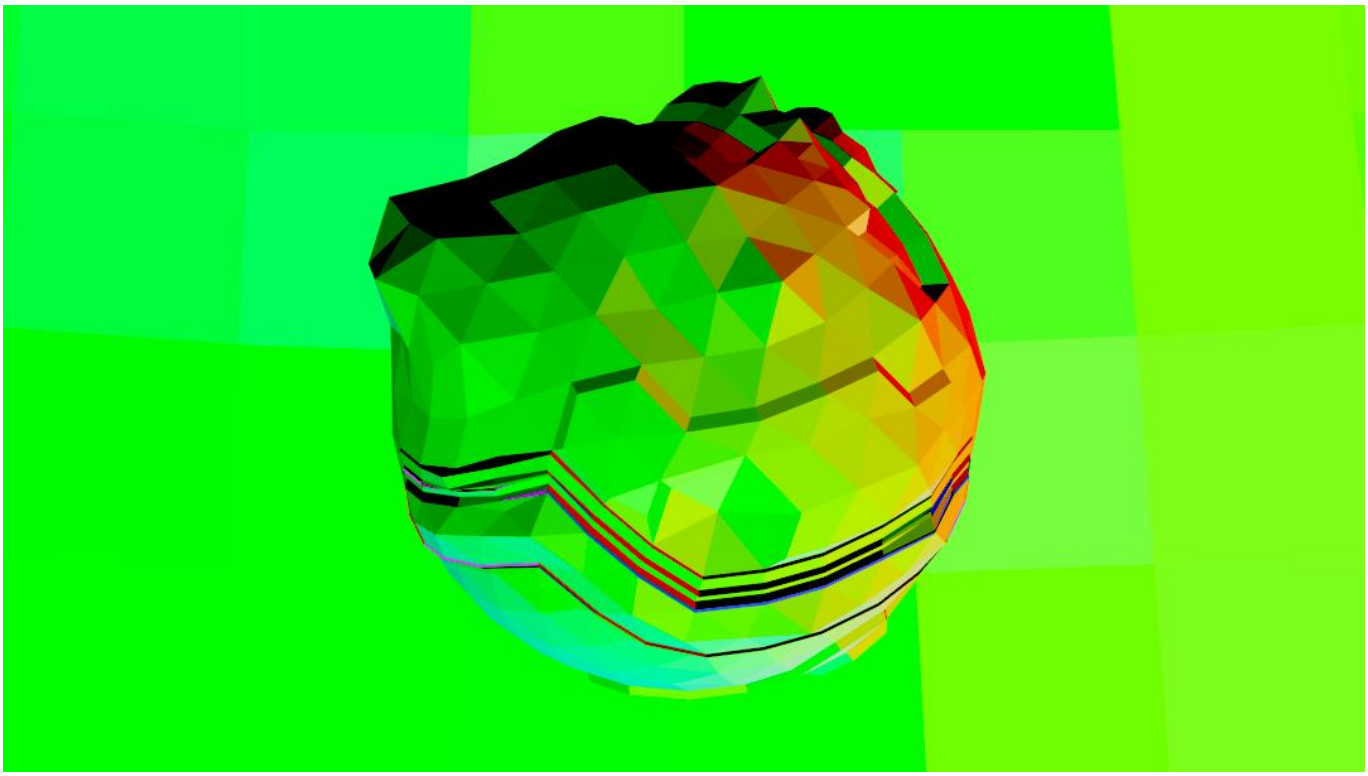
```
    const auto& instance = scene.instances[isec.instance];
    const auto& shape    = scene.shapes[instance.shape];
    const auto& material = scene.materials[instance.material];
    const auto& normal   = eval_normal(shape, isec.element, isec.uv);

    auto position = transform_point(
        instance.frame, eval_position(shape, isec.element, isec.uv));
    vec4f color = rgb_to_rgba(material.color);
    vec4f nor   = {normal.x, normal.y, normal.z, 0};
    vec4f ray4d = {ray.d.x, ray.d.y, ray.d.z, 0};

    //Comment one of the two code blocks

    //Normal Vector
    auto modx = fmod(position.x, 1);
    auto mody = fmod(position.y, 1);
    auto modz = fmod(position.z, 1);

    auto smooth = dot(normal, -ray.d);

    return vec4f{modx, mody, modz, 0} * (smooth + 0.5);

    //Position vector
    auto modx = fmod(normal.x, 1);
    auto mody = fmod(normal.y, 1);
    auto modz = fmod(normal.z, 1);

    return {modx, mody, modz, 0};

}
```

Le immagini ottenute sono le seguenti (1-2. position, 3-4.normal):









Cyber shader

Medio

L'obiettivo di questo algoritmo è realizzare dei materiali che possano ricordare gli ologrammi. Per raggiungere tale scopo il codice utilizza riflessione e trasparenza in modo randomico. Questo tipo di implementazione permette di variare, assegnando un valore di randomicità, l'influenza di queste due proprietà sul materiale risultante. Inoltre, l'approccio randomico genera dei disturbi sulle superfici interessate, restituendo il *noise* di una comunicazione o di una riproduzione in realtà aumentata.

Per realizzare questo *shader* sono state utilizzate delle reference quali:

- *Far Cry 3: Blood Dragon*
- *The Ascent*
- *Cyberpunk 2077*

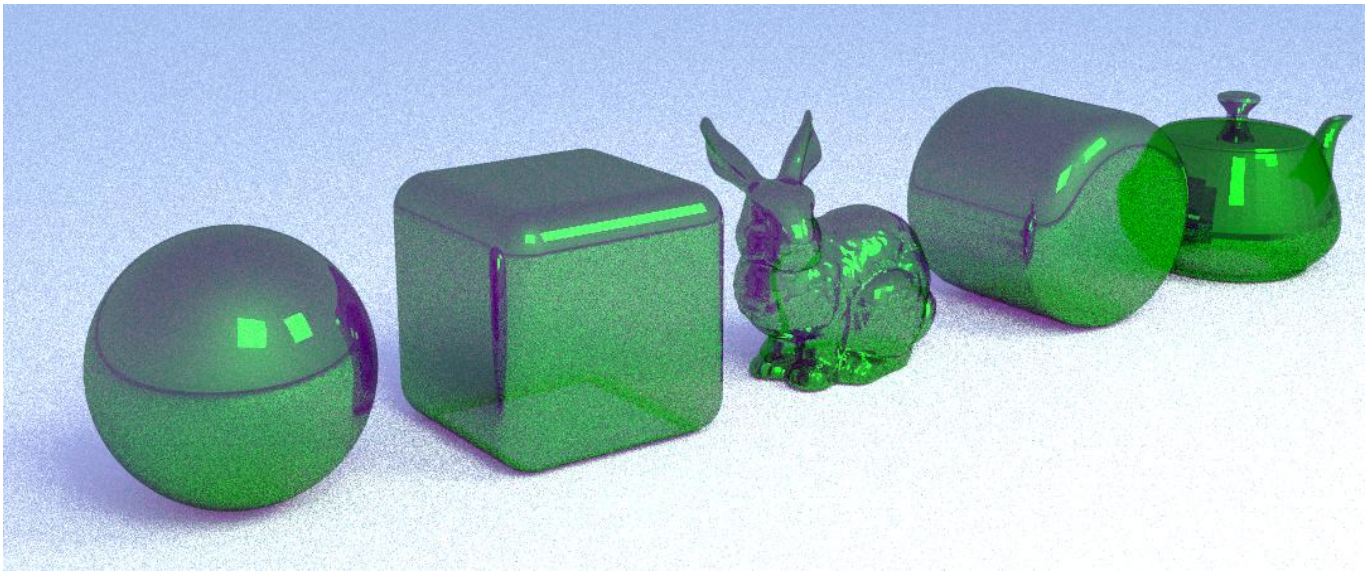
```
tatic vec4f shade_cyber(const scene_data& scene, const bvh_scene& bvh,
    const ray3f& ray, int bounce, rng_state& rng,
    const raytrace_params& params) {
  auto isec = intersect_bvh(bvh, scene, ray);
  if (!isec.hit) {
    vec3f res = eval_environment(scene, ray.d);
    return rgb_to_rgba(res);
  }

  const auto& instance = scene.instances[isec.instance];
  const auto& shape    = scene.shapes[instance.shape];
  const auto& material = scene.materials[instance.material];
  auto        normal   = eval_normal(shape, isec.element, isec.uv);

  auto position = transform_point(
      instance.frame, eval_position(shape, isec.element, isec.uv));
  vec4f color = rgb_to_rgba(material.color);
  vec4f nor   = {normal.x, normal.y, normal.z, 0};
  vec4f ray4d = {ray.d.x, ray.d.y, ray.d.z, 0};

  auto ret = rgb_to_rgba(material.emission);

  if (bounce >= params.bounces) {
    return ret;
  }

  switch (material.type) {
    case material_type::matte: {
      auto incoming = sample_hemisphere(normal, rand2f(rng));
      ret += (2 * M_PI) * color / M_PI *
             shade_cyber(scene, bvh, ray3f{position, incoming}, bounce + 1,
rng,
                 params) *
             dot(normal, incoming);
      return ret;
    }
```

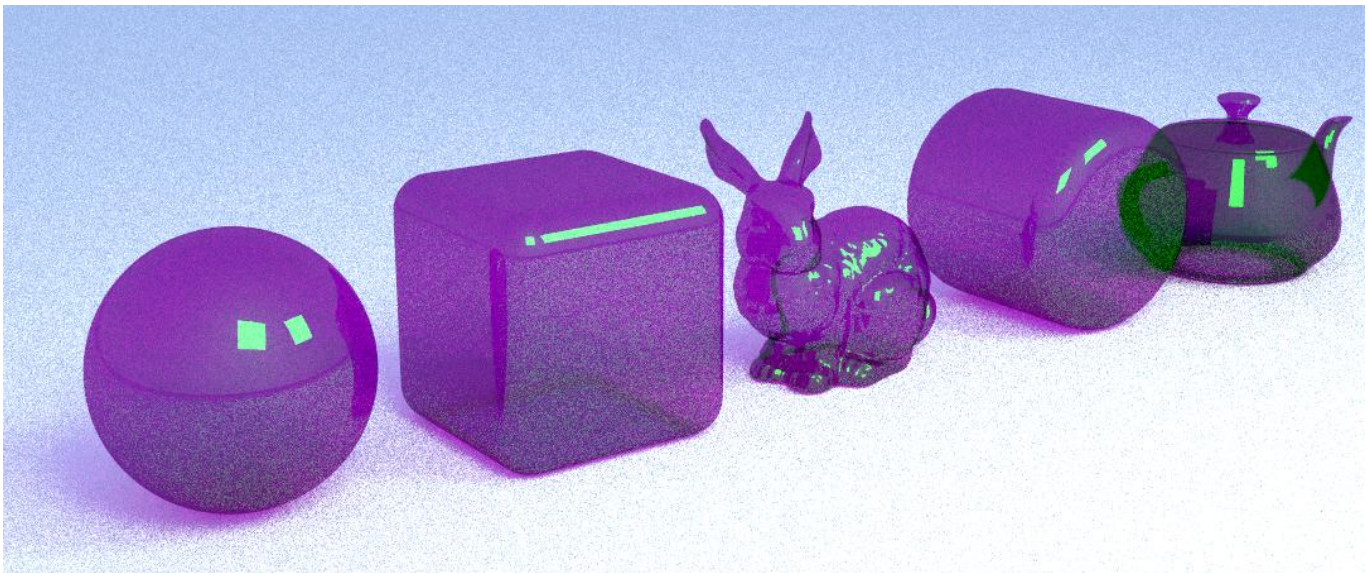
In quest'ultima parte l'algoritmo applica il reale effetto dello shader, generando randomicamente materiali trasparenti o riflettenti. Il codice genera queste due proprietà scegliendo tra due tonalità: viola e verde, la riflessione e la trasparenza possono essere regolate variando il livello di randomicità nella variabile *lower_bound*.

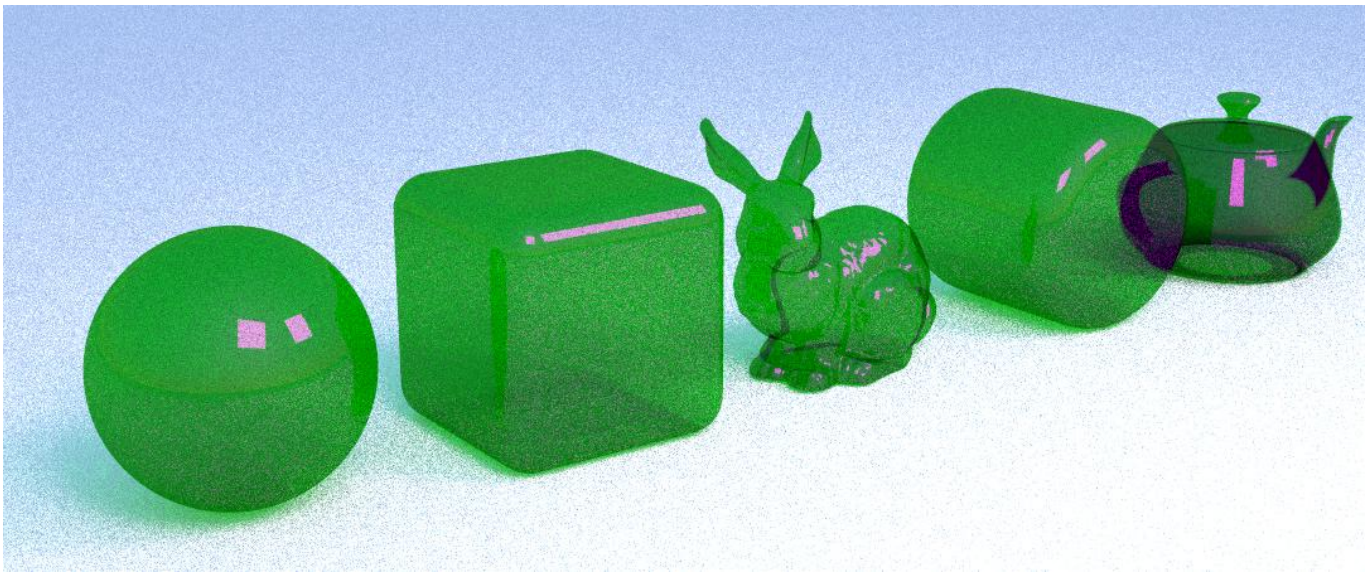
```
    case material_type::volumetric: {
      auto  randomizer  = rand1f(rng);
      float lower_bound = 0.05;
      if (randomizer > lower_bound) {
        ret += vec4f{0, 0.7, 0, 0} * shade_cyber(scene, bvh,
                                         ray3f{position, ray.d}, bounce +
1,
                                         rng, params);
```

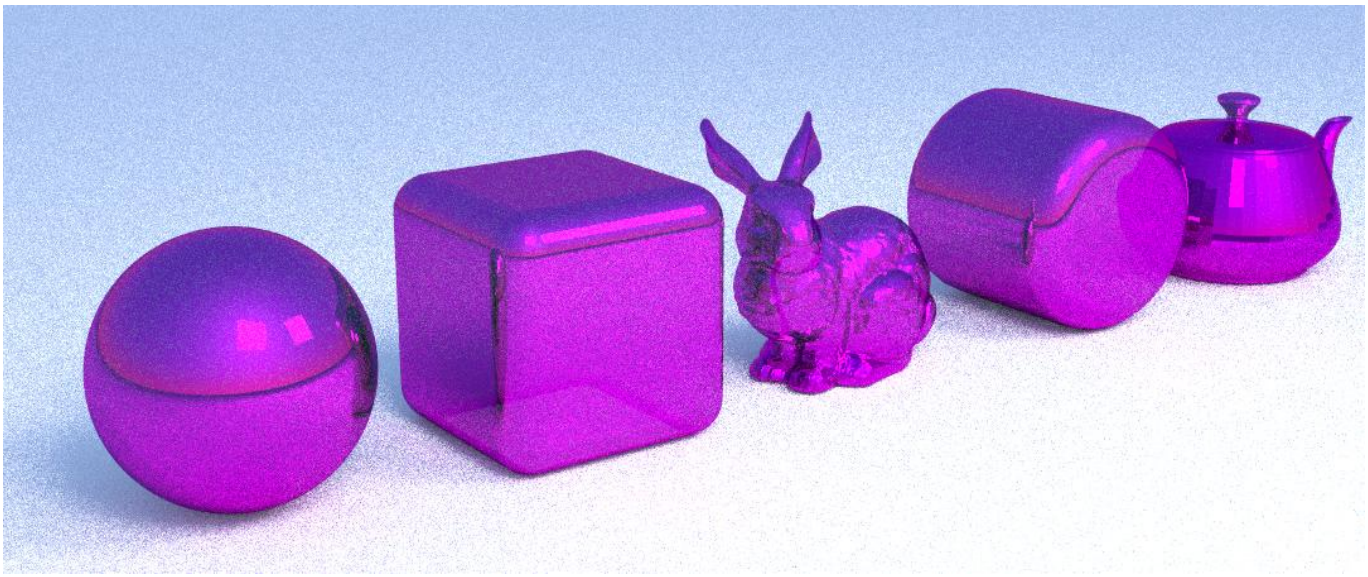
```
        } else {
          auto reflected = reflect(-ray.d, normal);
          ret += vec4f{0.7, 0, 0.7, 0} * shade_cyber(scene, bvh,
                                                ray3f{position, reflected},
                                                bounce + 1, rng, params);
        }
        return ret;
      }
    }
  }
```

Sono stati ottenuti risultati differenti intercambiando le due tonalità e variando il livello di randomicità, sono riportati di seguito i più significativi.









Stripes shader

Medio

In questo paragrafo è trattato un algoritmo che genera materiali metallici e riflettenti in intorni regolari. La manipolazione dei dati del punto d'intersezione, permette di effettuare dei controlli che alterano la percezione della superficie. In questo caso, sfruttando la posizione d'intersezione, sono generate delle parti del materiale completamente riflettenti ed altre completamente trasparenti. La percezione che si ha del volume, dopo aver effettuato il processo di *shading*, è che sia composto da più volumi indipendenti che, sovrapposti, realizzano una figura.

A differenza degli algoritmi descritti in precedenza, quest'ultimo manipola il colore assegnato al modello e non lo modifica nel codice.

```
static vec4f shade_stripes(const scene_data& scene, const bvh_scene& bvh,
    const ray3f& ray, int bounce, rng_state& rng,
    const raytrace_params& params) {
  auto isec = intersect_bvh(bvh, scene, ray);
  if (!isec.hit) {
```

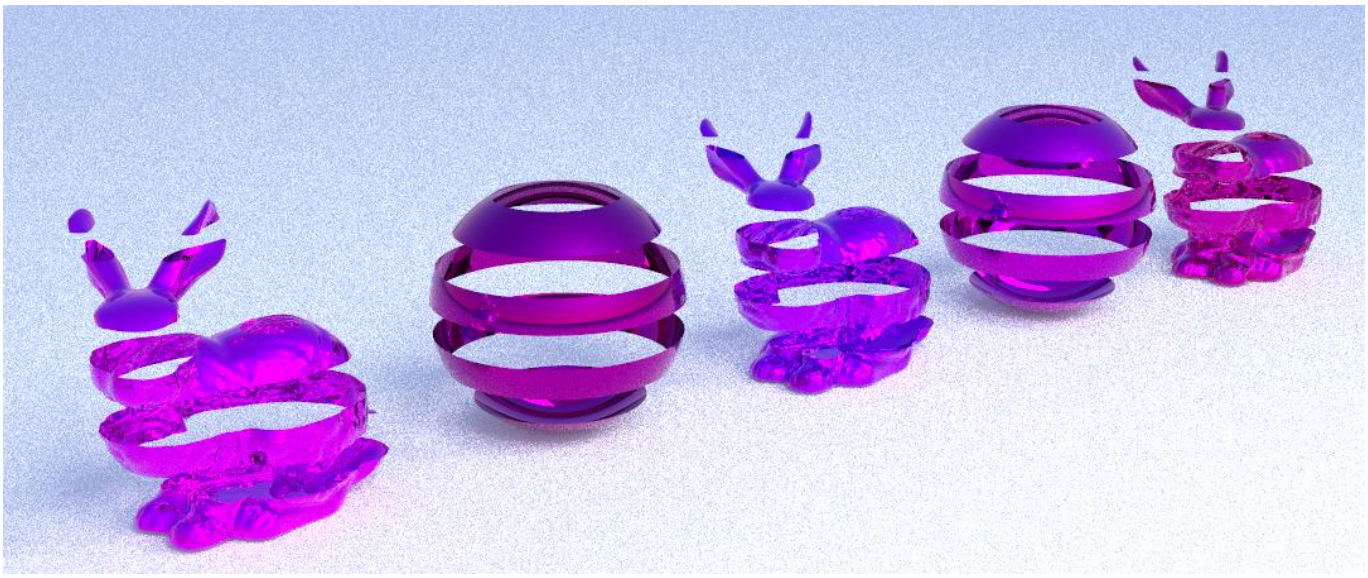
```
    vec3f res = eval_environment(scene, ray.d);
    return rgb_to_rgba(res);
  }

  const auto& instance = scene.instances[isec.instance];
  const auto& shape    = scene.shapes[instance.shape];
  const auto& material = scene.materials[instance.material];
  auto        normal   = eval_normal(shape, isec.element, isec.uv);

  auto position = transform_point(
      instance.frame, eval_position(shape, isec.element, isec.uv));
  vec4f color = rgb_to_rgba(material.color);
  vec4f nor   = {normal.x, normal.y, normal.z, 0};
  vec4f ray4d = {ray.d.x, ray.d.y, ray.d.z, 0};

  auto ret = rgb_to_rgba(material.emission);

  if (bounce >= params.bounces) {
    return ret;
  }

  switch (material.type) {
    case material_type::matte: {
      auto incoming = sample_hemisphere(normal, rand2f(rng));
      ret += color * shade_stripes(scene, bvh, ray3f{position, incoming},
                        bounce + 1, rng, params);
      return ret;
    }
```

Nella seconda parte, il codice genera i materiali d'interesse. Per suddividere le immagini in intervalli regolari, il codice effettua dei controlli sul valore della posizione d'intersezione sull'asse y. Se la posizione in y è compresa nell'intorno definito, allora la superficie verrà trattata come un materiale riflettente, altrimenti verrà trattata come un materiale trasparente senza alcuna riflessione nè densità.

```
    case material_type::reflective: {
      auto  reflected = reflect(-ray.d, normal);
      vec3f fresnel   = fresnel_schlick(material.color, normal, ray.d);
      vec4f fre       = rgb_to_rgba(fresnel);
      if (position.y >= 0 && position.y <= 0.02) {
        ret += fre * shade_stripes(scene, bvh, ray3f{position, reflected},
                          bounce + 1, rng, params);
      } else if (position.y >= 0.04 && position.y <= 0.06) {
        ret += fre * shade_stripes(scene, bvh, ray3f{position, reflected},
                          bounce + 1, rng, params);

      } else if (position.y >= 0.08 && position.y <= 0.1) {
        ret += fre * shade_stripes(scene, bvh, ray3f{position, reflected},
                          bounce + 1, rng, params);
      } else if (position.y >= 0.12 && position.y <= 0.14) {
        ret += fre * shade_stripes(scene, bvh, ray3f{position, reflected},
                          bounce + 1, rng, params);
```

```
        } else if (position.y >= 0.15 && position.y <= 0.18) {
          ret += fre * shade_stripes(scene, bvh, ray3f{position, reflected},
                         bounce + 1, rng, params);
        } else {
          ret += shade_stripes(
              scene, bvh, ray3f{position, ray.d}, bounce + 1, rng, params);
        }
        return ret;
      }
    }
  }
```

Di seguito sono riportati i risultati ottenuti dall'algoritmo applicando alla scena tonalità differenti.



Weird normal reflection shader

Medio

Lo shader realizzato in questo paragrafo non tenta di ricreare alcun materiale naturale, bensì genera superfici con riflessioni e colori che riportano a materiali per lo più onirici. Sfruttando riflessioni ribaltate, e

sfumature randomiche del colore assegnato l'algoritmo genera dei materiali simili a metalli liquidi.

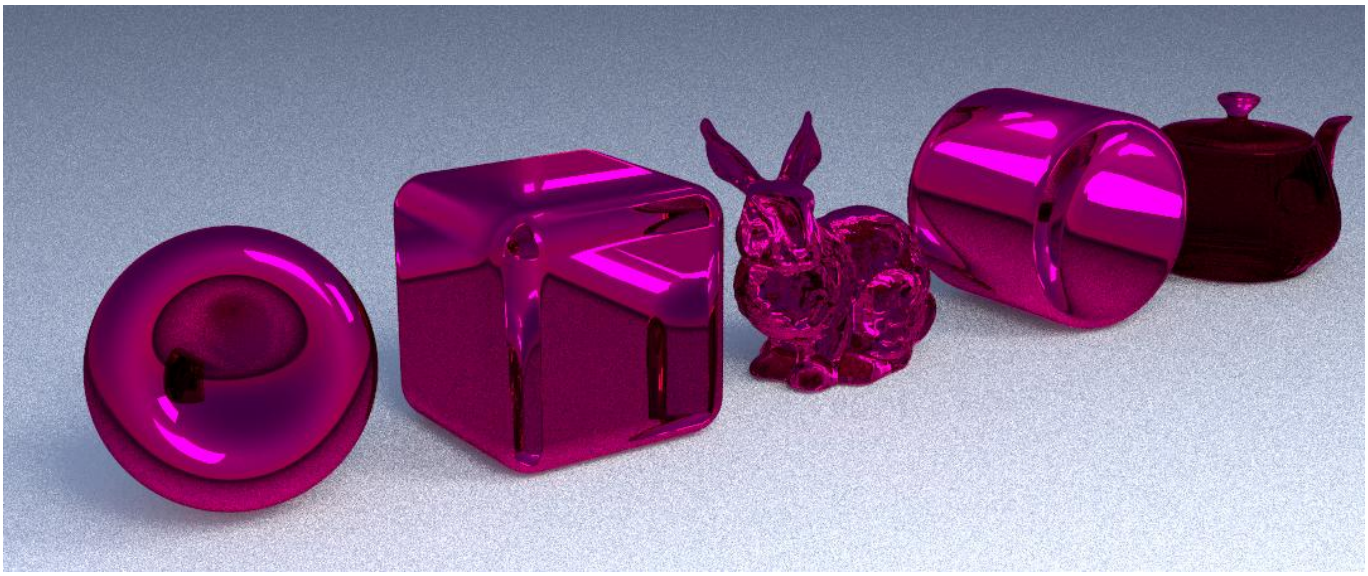
```
static vec4f shade_weird(const scene_data& scene, const bvh_scene& bvh,
    const ray3f& ray, int bounce, rng_state& rng,
    const raytrace_params& params) {
  auto isec = intersect_bvh(bvh, scene, ray);
  if (!isec.hit) {
    vec3f res = eval_environment(scene, ray.d);
    return rgb_to_rgba(res);
  }

  const auto& instance = scene.instances[isec.instance];
  const auto& shape    = scene.shapes[instance.shape];
  const auto& material = scene.materials[instance.material];
  auto        normal   = eval_normal(shape, isec.element, isec.uv);

  auto position = transform_point(
      instance.frame, eval_position(shape, isec.element, isec.uv));
  vec4f color = rgb_to_rgba(material.color);
  vec4f nor   = {normal.x, normal.y, normal.z, 0};
  vec4f ray4d = {ray.d.x, ray.d.y, ray.d.z, 0};

  auto ret = rgb_to_rgba(material.emission);

  if (bounce >= params.bounces) {
    return ret;
  }

  switch (material.type) {
    case material_type::matte: {
      auto incoming = sample_hemisphere(normal, rand2f(rng));
      ret += (2 * M_PI) * color / M_PI *
             shade_weird(scene, bvh, ray3f{position, incoming}, bounce + 1,
rng,
                 params) *
             dot(normal, -ray.d);
      dot(normal, -ray.d);
      return ret;
    }
```

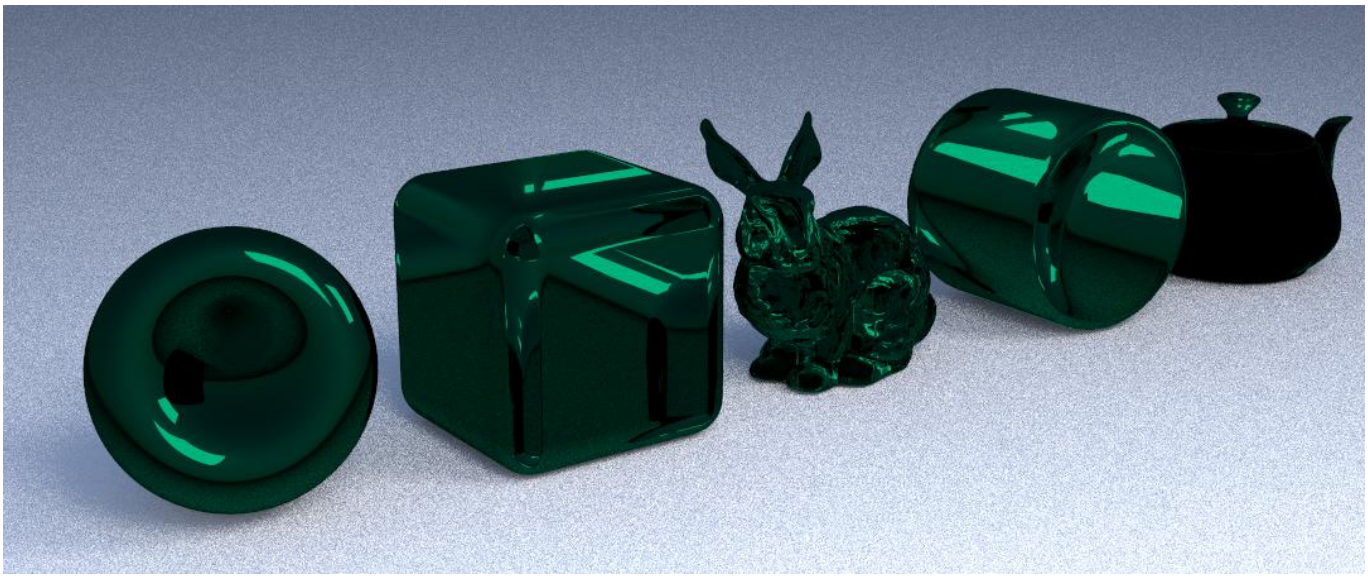
In quest'ultima fase lo shader genera le vere e proprie figure di interesse. Infatti, crea un vettore riflessione (*next*) che, utilizzando i vettori direzione e normale, genera degli effetti completamente innaturali e creativi. I modelli renderizzati saranno colorati da varie sfumature del colore assegnato in precedenza, tali sfumature saranno completamente randomizzate per ogni intersezione avvenuta.

```
    case material_type::volumetric: {
      auto red = 1;
      auto green = 0;
      auto blue = 0.5f;
      auto  next        = reflect(ray.d, normal);
```

```
        auto  randomizer  = rand1f(rng);
        float lower_bound = 0.05;
        ret += vec4f{red * rand1f(rng), green * rand1f(rng), blue *
rand1f(rng), 0} *
               shade_weird(
                   scene, bvh, ray3f{position, next}, bounce + 1, rng,
params);

        return ret;
      }
    }
}
```

Di seguito sono riportati i risultati ottenuti variando la tonalità di colore.





Sand & Dust shader

Medio

In questo paragrafo viene presentato un algoritmo che tenta di riprodurre materiali come sabbia e polvere. Per rappresentarli lavora sulla granulosità e la riflessione; Per quanto concerne la prima, il codice aumenta fortemente il *noise* nei materiali. Per la seconda utilizza la ricorsività della funzione per riflettere raggi randomici e riportarli sui materiali interessati, così facendo ogni raggio che intercetta i meteriali *glossy* viene riflesso secondo una generazione randomica.

A differenza degli algoritmi descritti in precedenza, quest'ultimo manipola il colore assegnato al modello e non lo modifica nel codice.

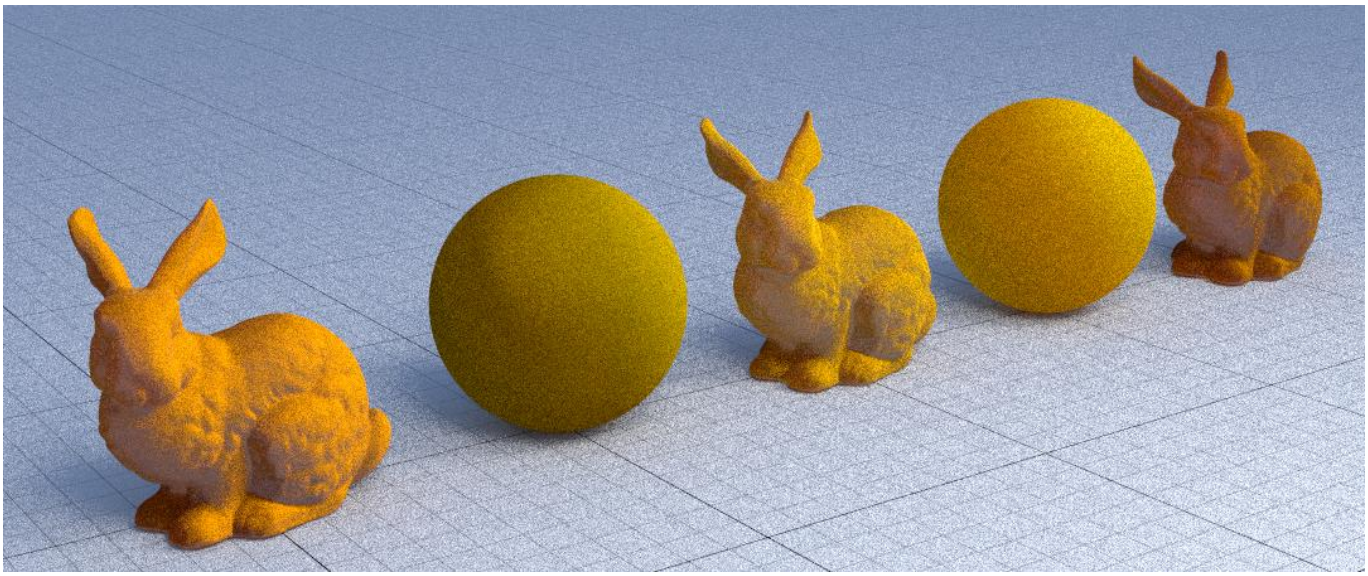
```
static vec4f shade_sand(const scene_data& scene, const bvh_scene& bvh,
    const ray3f& ray, int bounce, rng_state& rng,
    const raytrace_params& params) {
  auto isec = intersect_bvh(bvh, scene, ray);
  if (!isec.hit) {
    vec3f res = eval_environment(scene, ray.d);
    return rgb_to_rgba(res);
  }

  const auto& instance = scene.instances[isec.instance];
  const auto& material = scene.materials[instance.material];
  const auto& shape    = scene.shapes[instance.shape];

  auto normal = transform_direction(
      instance.frame, eval_normal(shape, isec.element, isec.uv));
  auto position = transform_point(
      instance.frame, eval_position(shape, isec.element, isec.uv));
  auto  texcoord = eval_texcoord(shape, isec.element, isec.uv);
  auto  mat      = eval_material(scene, instance, isec.element, isec.uv);
  auto  texture  = eval_texture(scene, material.color_tex, texcoord);
  vec4f color    = rgb_to_rgba(mat.color);

  auto  radiance = material.emission;
  vec4f rad      = rgb_to_rgba(radiance);

  if (bounce >= params.bounces) {
    return rad;
  }

  if (!shape.points.empty()) {
    normal = -ray.d;
  } else if (!shape.lines.empty()) {
    normal = orthonormalize(-ray.d, normal);
  } else if (!shape.triangles.empty()) {
    if (dot(-ray.d, normal) < 0) {
      normal = -normal;
    }
  }

  switch (material.type) {
    case material_type::matte: {
      auto incoming = sample_hemisphere(normal, rand2f(rng));
      rad += (2 * M_PI) * color / M_PI *
```

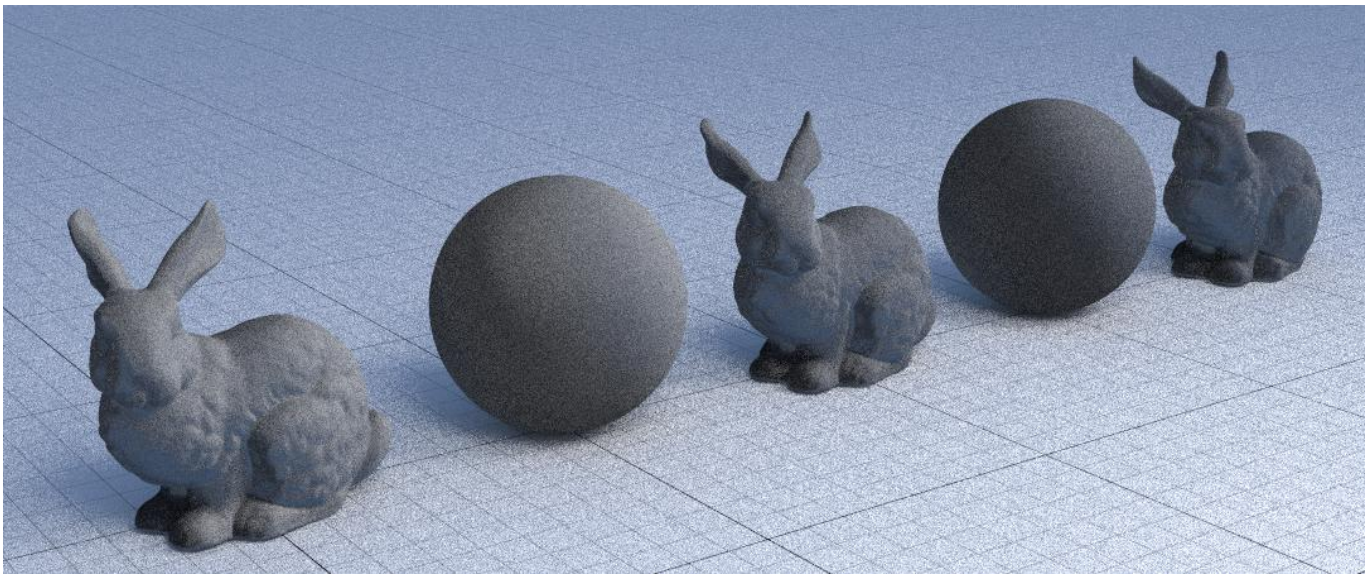
```
                shade_sand(scene, bvh, ray3f{position, incoming}, bounce + 1,
rng,
                    params) *
                dot(normal, incoming);
        return rad;
      }
```

L'algoritmo genera i materiali sabbiosi in quest'ultima fase. Per ottenere tale risultato genera un vettore *reflection* il più randomico possibile per simulare la riflessione dei granuli: la creazione di tale vettore viene effettuata utilizzando il vettore *halfway*, il quale dipende fortemente dalla *roughness* del materiale, e da un vettore randomico, utile per creare rimbalzi casuali. Infine, il colore viene resituito manipolato dall'approssimazione di *Fresnel Schlick* per migliorare la riflessione sull'intero materiale.

```
      case material_type::glossy: {
        auto  exponent = M_PI / pow(mat.roughness, M_PI * M_PI);
        auto  halfway = sample_hemisphere_cospower(exponent, normal,
rand2f(rng));

        vec3f randvec = rand3f(rng);
        auto  reflection = reflect(halfway, randvec);

        auto  fresnel  = fresnel_schlick(mat.color * 0.8, randvec, halfway);
        vec4f fre      = rgb_to_rgba(fresnel);

        rad += fre * shade_sand(scene, bvh, ray3f{position, reflection},
bounce + 1,
                        rng, params);
        return rad;

        return rad;
      }
    }
  }
```

Questo è il risultato ottenuto su materiali di tonalità arancione e grigia, per la riproduzione della sabbia e della polvere.





Cel-shader

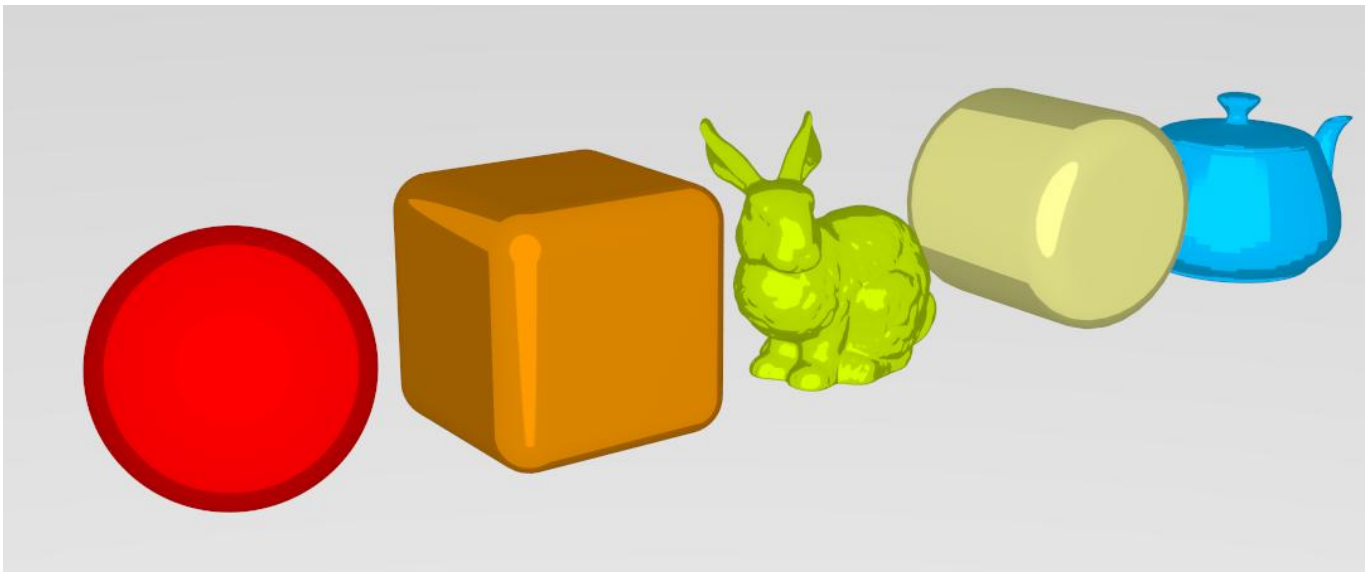
Medio

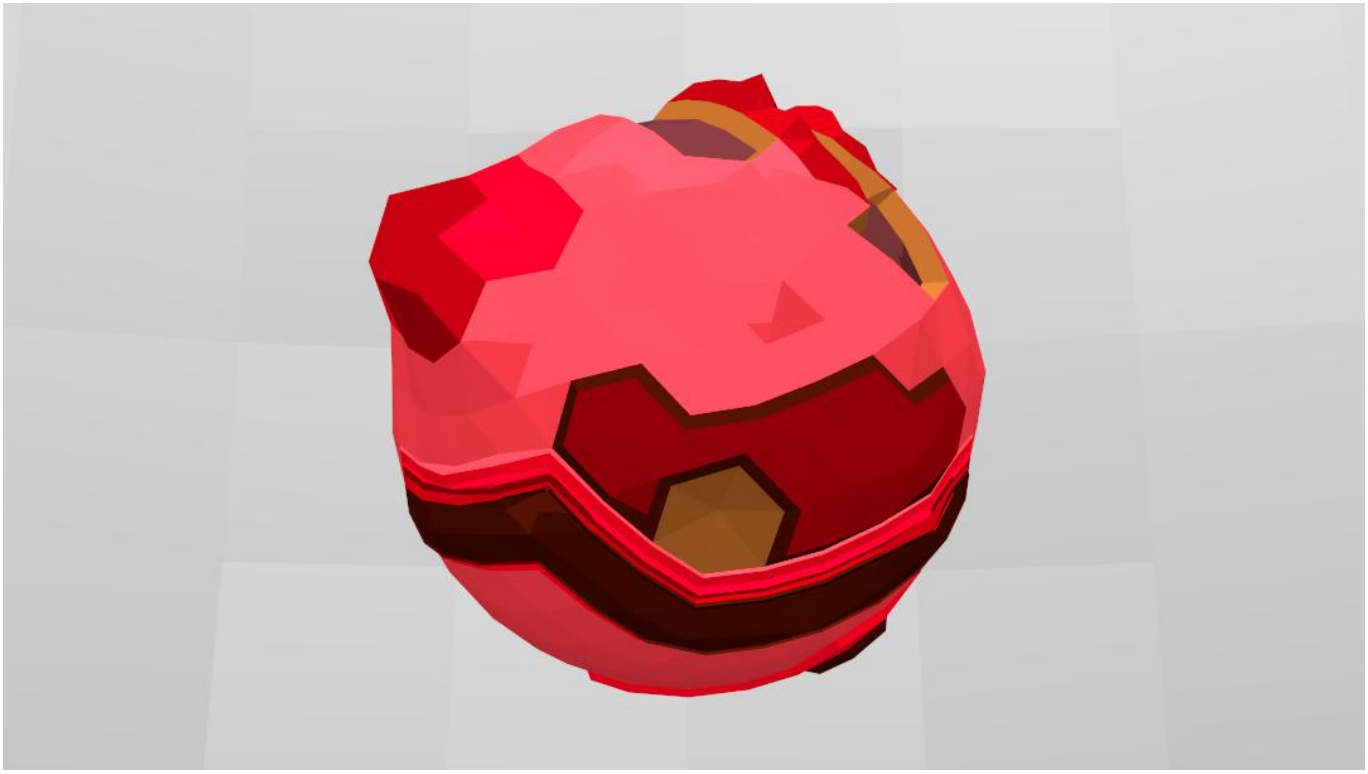
Il codice riportato in questa sezione, effettuando delle manipolazioni sui dati ottenuti dall'intersezione dei raggi con i materiali, restituisce un scena dove vengono rimossi tutti i fading tra punti luce, colore e ombre. I colori risultanti appaiono uniformati e separati da ombre e luci.

```
static vec4f shade_toon(const scene_data& scene, const bvh_scene& bvh,
    const ray3f& ray, int bounce, rng_state& rng,
    const raytrace_params& params) {
  auto isec = intersect_bvh(bvh, scene, ray);
  if (!isec.hit) {
    return {0, 0, 0, 0};
  }

  const auto& instance = scene.instances[isec.instance];
  const auto& shape    = scene.shapes[instance.shape];
  const auto& material = scene.materials[instance.material];
```

```
    const auto& normal   = eval_normal(shape, isec.element, isec.uv);

    auto position = transform_point(
        instance.frame, eval_position(shape, isec.element, isec.uv));
    vec4f color = rgb_to_rgba(material.color);
    vec4f nor   = {normal.x, normal.y, normal.z, 0};
    vec4f ray4d = {ray.d.x, ray.d.y, ray.d.z, 0};

    auto incoming = sample_hemisphere(normal, rand2f(rng));
    auto NdotL    = dot(nor, -ray4d);

    vec4f envColor = {0.4, 0.4, 0.4, 1};
    vec4f rimColor = {1, 1, 1, 1};
    vec4f speColor = {0.9, 0.9, 0.9, 1};
    auto  light    = 0;
    if (NdotL > 0.5) {
      light = 1;
    } else {
      light = 0;
    }

    vec3f viewDir                = normalize(ray.d);
    vec3f halfVector             = normalize(ray.d + position);
    float NdotH                  = dot(normal, halfVector);
    float specularIntensity      = pow(NdotL * light, 8 * 8);
    float specularIntensitySmooth = smoothstep(0.005, 0.01,
  specularIntensity);
    vec4f specular               = specularIntensitySmooth * speColor;

    float rimDot       = (1 - dot(viewDir, normalize(normal))) / 2;
    float rimIntensity = smoothstep(0.716 - 0.01, 0.716 + 0.01, rimDot *
  NdotL);
    vec4f rim          = rimIntensity * rimColor;

    return color * (light + envColor + specular + rimDot);
  }
```

I risultati ottenuti sono i seguenti:





Extra renders

In conclusione sono riportati altri render effettuati con gli algoritmi descritti in precedenza nella sezione **Experiments**. Tutte le immagini prodotte sono realizzate con una scena e tre modelli del gioco ***Gravio***, modellati e assemblati su **Blender**.