Rapport

Aspects théoriques de l’informatique graphique

Partie rendu

Gauthier Bouyjou

2. Estimation des statistiques par pixel

Q1:

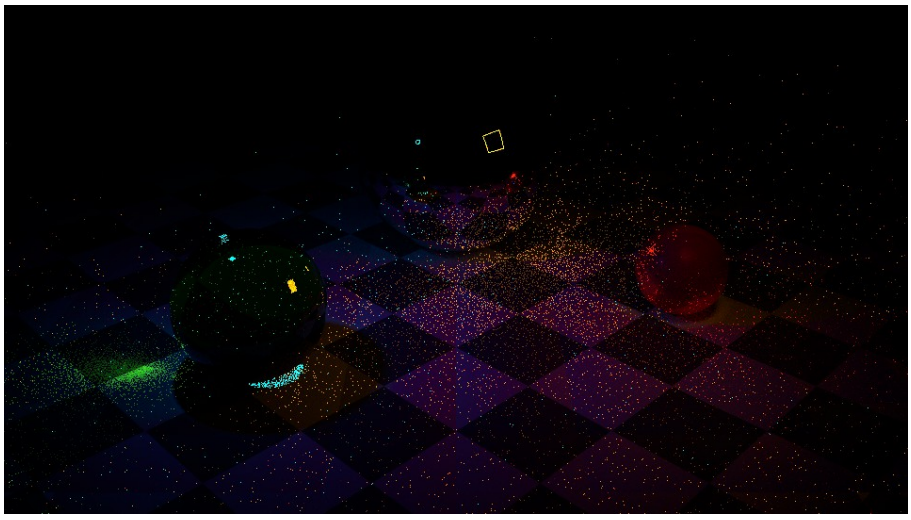
Implémentation de l’algorithme de Welford (online) dans la méthode UpdateStats :

```
88  void Statistics::UpdateStats(Point2i pixel, Spectrum &&L) {
89      if (!InsideExclusive(pixel, pixelBounds))
90          return;
91
92      Pixel &statsPixel = GetPixel(pixel);
93      long &samples = statsPixel.samples;
94      Spectrum &mean = statsPixel.mean;
95      Spectrum &moment2 = statsPixel.moment2;
96
97      // Welford's online algorithm
98      ++samples;
99      auto delta = L - statsPixel.mean;
100     mean += delta / samples;
101     auto delta2 = L - mean;
102     moment2 += delta * delta2;
103 }
```

Q2:

Calcul de la variance pour chaque échantillon :

```
Spectrum Statistics::Variance(const Pixel& statsPixel) const
{
    return statsPixel.moment2 / (statsPixel.samples - 1);
}
```



Les zones proches du noir correspondent à de faible variance, en général cela correspond aux zones faiblement éclairées (de faible énergie).

Les autres zones qui restent ont une variance proportionnelle à l'intensité de la luminance.
On peut remarquer qu'on distingue parfaitement les contours des éclairages directs dus à l'imprécision de chaque rayon envoyé depuis la source (suivant une loi uniforme) pour chaque échantillon d'un même pixel, certains de ces rayons intersectent la source de lumière (luminance forte) et d'autres passent à côté (luminance plus faible).
On se retrouve donc avec une très forte variance de ces deux groupes sur un même pixel.

Les zones de bruits aux sols sont dues à la possibilité de réflexion multiple avec les objets de la scène.

La caustique émise derrière la sphère verte génère une forte variance aux sols due aux alternances de rayons qui sont dispersés aux sols, il faudrait envoyer plus de rayons pour avoir une dispersion plus uniforme de ces rayons.

3. Définition et analyse de l'erreur

Q3 :
Pour un nombre d'échantillons infinis la variance devient nulle car le « path tracing » est non biaisé, et l'espérance de la variable est bien une constante, du coup à l'infini on tend vers 0.
Je ne traite pas le cas asymptotique avec un nombre d'échantillon nul (insensé).

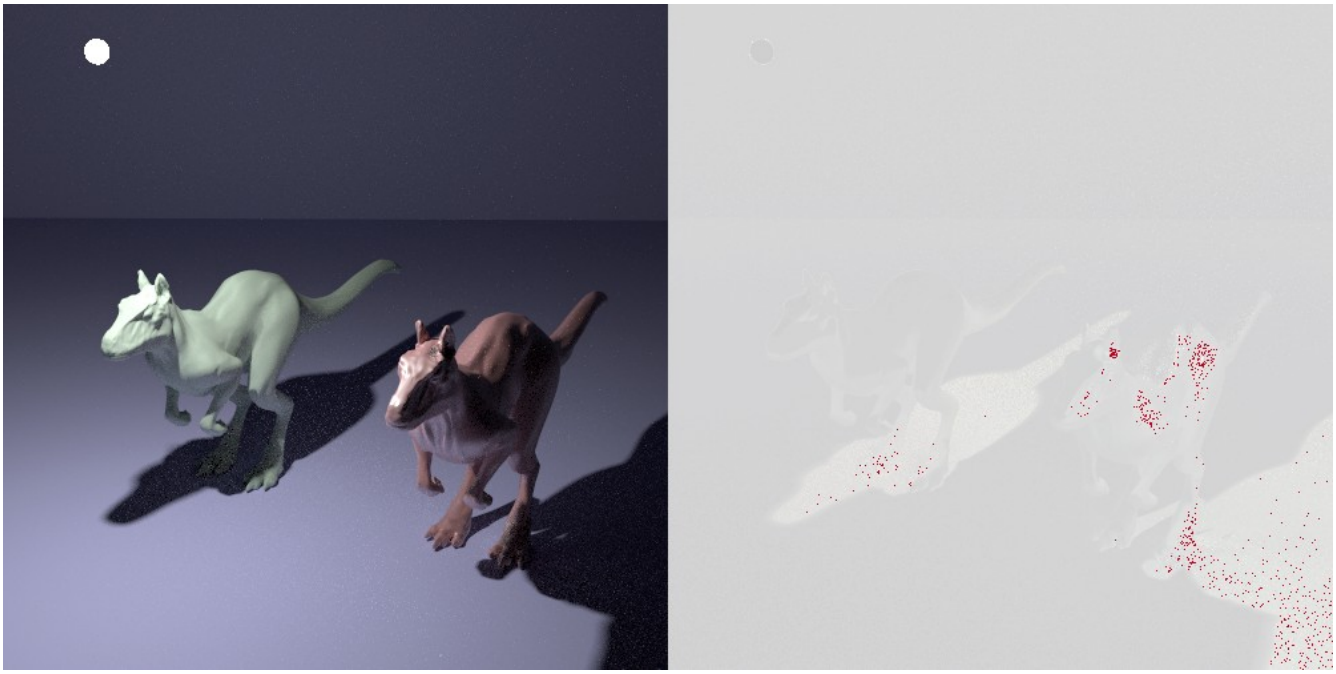
Q4 :

```
Spectrum Statistics::Error(const Pixel& statsPixel) const
{
    const Spectrum sigma_x = Sqrt(Variance(statsPixel)) / sqrt(statsPixel.samples);
    const auto& mean = statsPixel.mean;

    if (errorHeuristic != Heuristic::RELATIVE) {
        return sigma_x;
    }

    Float rgb[3];
    mean.ToRGB(rgb);
    const Float epsilon = 10e-4f;

    for (int i = 0; i < 3; ++i) {
        if (rgb[i] <= epsilon) {
            //            rgb[i] = 0.0f; // option A
            rgb[i] = epsilon; // option B

        } else {
            //            rgb[i] = sigma_x[i] / rgb[i]; // option A
        }
    }
    //    return pbrt::Spectrum::FromRGB(rgb); // option A
    return sigma_x / pbrt::Spectrum::FromRGB(rgb); // option B
}
```

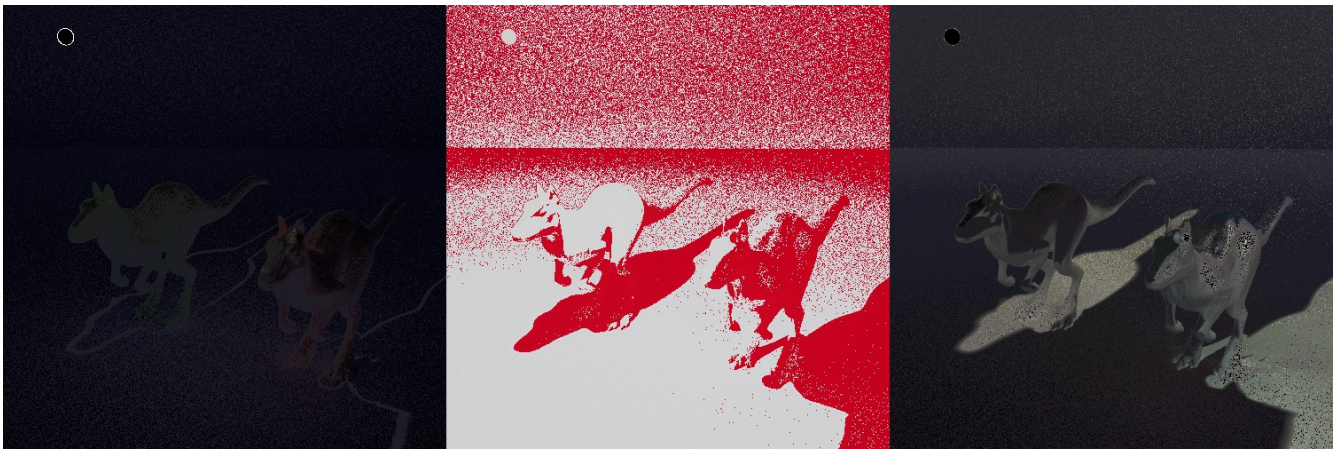
Option B : la plus naturelle
on « clamp » la moyenne [epsilon, ∞] (epsilon positif)
pour éviter la division par zéro, on aura donc pour des zones très sombres une erreur très grande si la variance est grande, et 0 si la variance tend vers 0.
On relève donc une plus forte erreur sur les zones à faible énergie, on risque du coup de lancer beaucoup d'échantillons sur des zones sombres diffuses, ce qui est dommage.
C'est pour cela que je propose une autre solution à ce problème.

Option A :
Si on a une moyenne proche de zéro, pixel noir, je suppose qu'il n'y a pas d'erreur possible car la moyenne est strictement positive, ce qui implique que la variance est très faible car une moyenne proche de zéro implique que tous les échantillons sont proches de zéro car il n'y a pas de luminance négative.
Donc je fixe une valeur d'erreur de 0, je n'applique pas la division par la moyenne sur cet élément là.

Comparaison option A et B, les pixels rouges sur la photo de droite correspondent à des pixels que j’ai fixé noir des que la moyenne était très faible et donc de dire qu’il n’y a pas d’erreur sur ce pixel.
On remarque que les pixels, qui ont été forcés au noir, sont dans des zones très sombres de base donc visuellement on ne voit pas de différence sur ces zones là.



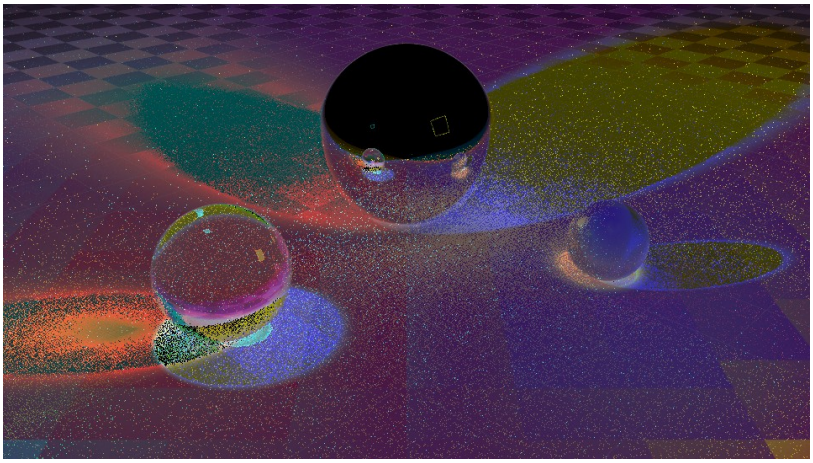
Q5 :
Erreur standard (gauche), différence des 2 erreurs (milieu), erreur relative (droite)



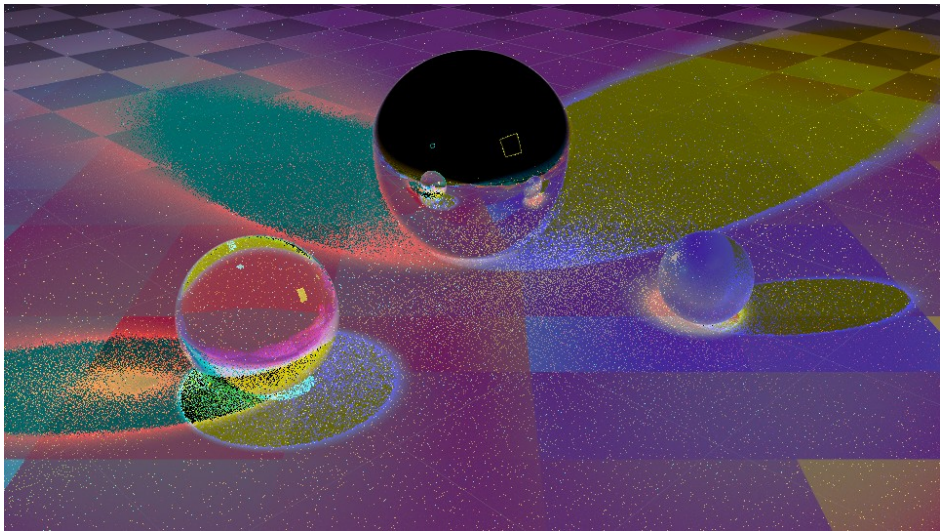
On remarque une différence majeure des 2 heuristiques, en effet la relative s’intéresse plus aux zones sombres, et néglige plus les zones de forte luminance où l’information

est peut être plus intéressante, zone de hautes fréquences, où on cherche un maximum de détails.
On peut remarquer que l'erreur relative exclue complètement l'erreur qui est due à l'aliasing au niveau du contour de la source de lumière de cette scène où la variance est maximale mais est atténuée par la moyenne de ce pixel du à une très grande luminance reçue (lumière directe).

Remarque erreur relative :
variance : erreur :



L'erreur étant liée à la variance, on remarque sur la photo de gauche que la variance sur des sources de lumière étendue (surfacique) suit une loi uniforme, d'où la présence de carré sur cette source.
Ce n'est en aucun cas lié à la taille des tuiles pour parallélisation lors de la boucle de rendu (en changeant la variable tileSize, la taille des carrés ne change pas).
Cela pourrait venir de comment pbrt stock la luminance dans la scène, peut être sous forme d'un octree, ou un quadrillage 3d régulier.
En doublant la taille des carreaux du damier au sol : Scale 20 20 20 → Scale 40 40 40, et une translation sur x de la texture, les carrés suivent. On peut conclure que c'est directement lié à la répétition des carreaux de texture, uv clampé entre [0, 1].



4. Contrôle adaptatif de la qualité

Q6.

Ajout clause Erreur dans SamplingLoop :

```
94          case Mode::ERROR:
95              // bootstrap
96              for (long i = 0; i < minSamples; ++i)
97                  loop();
98
99              // adaptive
100             count = minSamples;
101             while (!StopCriterion(pixel) && ++count < maxSamples)
102                 loop();
103             break;
```

Critère d'arrêt dans StopCriterion :

```
if (errorHeuristic != Heuristic::CONFIDENCE) {
    return dist(Error(statsPixel)) <= errorThreshold;
}
```

Pour la distance utilisée j'ai choisi premièrement la distance Euclidienne, mais vu que l'œil humain perçoit les couleurs avec une sensibilité différente pour chacune, j'ai opté par une égalisation avec des poids (fonction y() déjà présente dans Spectrum).

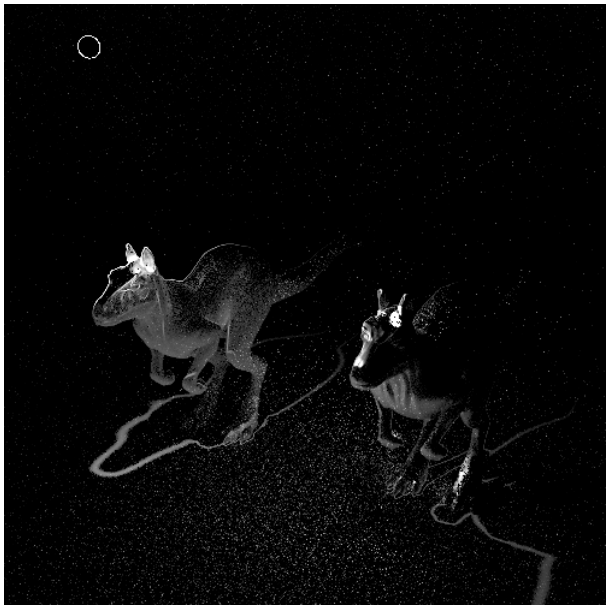
```
Float dist(const Spectrum& s)
{
    //    return sqrt(s[0] * s[0] + s[1] * s[1] + s[2] * s[2]); // Euclidean
    return s.y();
}
```

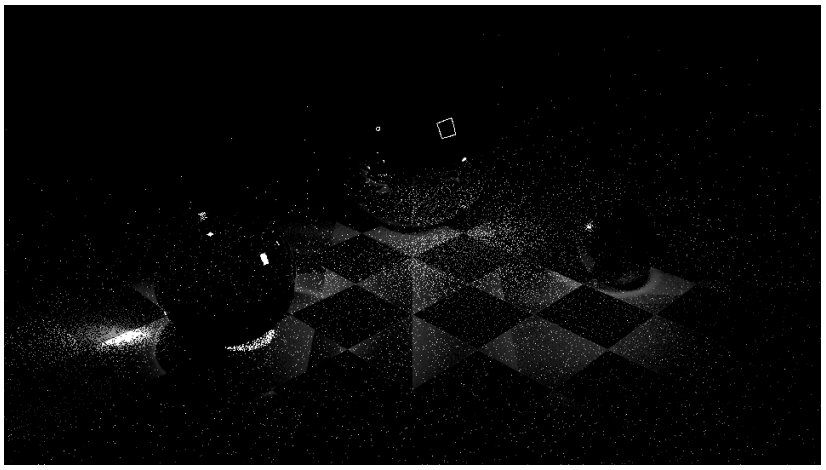
Q7.

```
Float Statistics::Sampling(const Pixel& statsPixel) const
{
    switch (mode) {
    case Mode::ERROR:
        return (static_cast<Float>(statsPixel.samples) - minSamples) / (maxSamples - minSamples);

    case Mode::TIME:
    case Mode::NORMAL:
        return 0.0f;
    }
}
```

On normalise (changement d’échelle) le nombre d’échantillon calculé pour chaque pixel pour une visualisation en nuance de gris.

En utilisant l’erreur standard :





Globalement on sur-échantillonne sur les zones de hautes fréquences (géométrie du modèle, contour direct avec fort contraste) , ou zones de diffraction de lumières (fractale, rebondissement de rayon sur surface mate).

5. Comparaison avec un rendu en temps fixe

Q8.

Ajout dans StartNextBatch la clause :

```
bool Statistics::StartNextBatch(int number)
{
    switch (mode) {
    case Mode::TIME:
        return ElapsedMilliseconds() < targetSeconds * 1000;

    default:
        if (batchOnce) {
            batchOnce = false;
            return true;
        } else
            return false;
    }
}
```

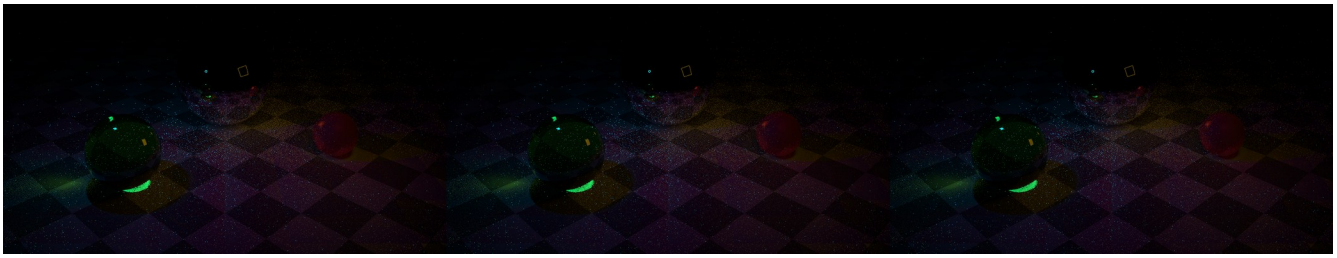
Ajout dans SamplingLoop la clause :

```
case Mode::TIME:
    for (uint i = 0; i < std::min(batchSize, maxSamples); ++i)
        loop();
}
```

On tire un nombre fixe de batchSize échantillons sans dépasser maxSamples par lot.

Q9.
Comparaison du nombre de batch pour une durée de 30 sec (targetseconds = 30s)

Comparaison erreurs standard :
gauche : batchsize = 1 milieu : batchsize = 16 milieu : batchsize = 64



On remarque que avec un batchsize de 1 on a plus d'erreurs, mais c'est sûrement du au manque d'imprécision sur le temps d'arrêt, avec un batch plus petit on est plus précis en temps.
Mais cela reste très faible.
La modification de la graine dans la boucle de rendu permettrait d'être plus aléatoire.

Q10.
Pour comparer deux méthodes, le critère d'égalité sera le temps d'exécution des deux méthodes, on pourra donc par la suite comparer le taux d'erreurs des deux images.
Et comparer les images rendues avec des images de références (fournit sur le site de pbrt) qui ont été calculées avec d'autres intégrateurs et avec beaucoup plus d'échantillons (4096 échantillons par pixel en moyenne).

Le protocole étant de lancer en mode erreur en chronométrant le temps d'exécution, puis de lancer en mode time en fixant le temps d'exécution du mode erreur.
En fixant le mode erreur en premier avec une heuristique standard, sur un nombre d'échantillon allant de 256 à 1024.
Les temps de calcul sont longs sur cpu, je décide donc d'automatiser cette tâche par un script bach (fourni dans l'archive de mon rendu de tp).
Pour chaque image fournie sur le site de pbrt, on doit modifier les fichiers pbrt pour intégrer notre intégrateur, on viendra modifier les variables membres de Statistics à la main dans le code si on souhaite changer des valeurs (maxSamples, minSamples, mode).

Après cela le script exécute notre exécutable pbrt pour chaque fichier pbrt, compare l'image de sortie avec l'image de référence correspondante, écrit dans le fichier pbrt correspondant le temps d'exécution de pbrt pour que lors d'une seconde passe avec le mode time on respecte le même temps de calcul pour chaque pbrt. Puis je rassemble toutes les photos calculées pour chaque mode dans une seule pour meilleure visualisation respectant un ordre/format précis qui est :

image de référence	différence ref/out	image calculée avec l'intégrateur adaptative
erreur standard	nombre d'échantillon	variance

nom du script : compareErrorTime.sh

deux passes :
on fixe à la main dans adaptive.cpp le membre mode à ERROR pour une première exécution, puis à TIME pour la deuxième exécution.

On peut facilement voir la performance des deux méthodes en regardant la différence avec l'image de référence de chacune des photos dans les deux répertoires error et time séparément.

Les photos résultantes vont être données à part dans l'archive dans le répertoire image. Les photos rendues vont suivre le format suivant :

Paramètres pbrt constants : minSamples = 32, errorThreshold = 0.001

maxSamples	error_standard	time	error_relative
256	2x6	2x6	2x6
1024	2x6	2x6	2x6
2048	2x6	2x6	2x6

Ou 2x6 représente le format de photo vu précédent (page précédente).
Pour une même photo on aura plusieurs résultats avec un lancé maximum d'échantillon de 256|1024|2048, qui permettra de voir si une méthode est plus gourmande/économe qu'une autre en nombre d'échantillon lancés.
Et puis on testera pbrt en mode error avec l'heuristique standard et relative, et le mode time avec un temps égale à celui du mode error_standard pour un même nombre maximum d'échantillons.

Example fichier bmw-m6 : (voir dossier images pour plus de précision)

En comparant les résultats entre erreur standard et time, on remarque que l'erreur standard se focalise plus vite sur les zones de fortes variances comparé au temps qui

travaille de manière uniforme sur toutes l'image (comparez le sol à droite de la voiture ou le par-choc avant, il apparaît blanc donc très peu d'erreur sur cette zone la, mais qui n'a pas grand intérêt puisque zone diffuse sans détail.

6. Intervalles de confiance

Q11 :

Reprise du code source de mitsuba : src/integrators/misc/adaptive.cpp

Ajout de variables membres dans la classe Statistics :

```
    // gauthier additions
    Float m_quantile;
    const Float m_pValue;
    ulong m_iSample;
    Spectrum m_averageLuminance;
```

Dans le constructeur de Statistics de adaptative on initialise le quantile :

```
    , m_pValue(0.05f)
    , m_iSample(0)
    , m_averageLuminance(0.0f)
{
    boost::math::normal dist(0, 1);
    m_quantile = static_cast<Float>(boost::math::quantile(dist, 1.0f - m_pValue / 2.0f));
```

Dans UpdateStats on met à jour la luminance moyenne de la scène (online comme la moyenne du pixel) :

```
    ++m_iSample;
    delta = L - m_averageLuminance;
    m_averageLuminance += delta / m_iSample;
```

Dans StopCriterion si l'heuristique est celle de « confidence », alors on utilise l'intervalle de confiance comme critère d'arrêt :

```
// Control approximation error
if (errorHeuristic != Heuristic::CONFIDENCE) {
    return dist(Error(statsPixel)) <= errorThreshold;
}

const long& samples = statsPixel.samples;
const Spectrum& mean = statsPixel.mean;
const Spectrum& variance = Variance(statsPixel);

auto stdError = Sqrt(variance / samples);

auto ciWidth = stdError * m_quantile;
auto base = max(mean, m_averageLuminance * 0.01f);

return ciWidth <= errorThreshold * base;
```

Utilisation d’une fonction max entre deux spectres :

```
Spectrum max(const Spectrum& left, const Spectrum& right)
{
    if (dist(left) > dist(right)) {
        return left;
    }
    return right;
}
```

image calculées avec un intervalle de confiance de 95 % (page suivante)
minSamples = 32
maxSamples = 1024
errorThreshold = 0.01