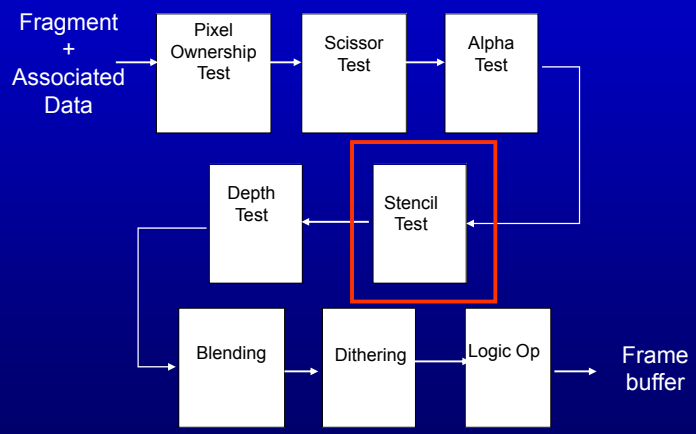
Stencil Buffer
Shadows Volumes

Xavier Décoret
Kadi Bouatouch

Per-Fragment Pipeline



Stencil Buffer

Alpha Testing

glAlphaFunc - specify the alpha test function

PARAMETERS

- *func* Specifies the alpha comparison function. Symbolic constants **GL_NEVER**, **GL_LESS**, **GL_EQUAL**, **GL_LEQUAL**, **GL_GREATER**, **GL_NOTEQUAL**, **GL_GEQUAL**, and **GL_ALWAYS** are accepted. The initial value is **GL_ALWAYS**.
- *ref* Specifies the reference value that incoming alpha values are compared to. The alpha test discards fragments depending on the outcome of a comparison between an incoming fragment's alpha value and a constant reference value.

Stencil Buffer

- Buffer auxilliaire non affiché
  - en plus des buffers de couleur et de profondeur
- Contrôle si un fragment réussit le test ou pas
  - spécification d'un *stencil test*

```
glStencilFunc(GL_EQUAL,0,~0);
```

- Modifié lors de la rasterisation
  - incrémentable/décrémentable
  - trois actions spécifiables
    - le fragment ne réussit pas le *stencil test*
    - le fragment réussit le *stencil test* et le réussit/rate le *z-test*
    - le fragment réussit à la fois le *stencil test* et le *z-test*

```
glStencilOp(GL_KEEP,GL_KEEP,GL_DECR);
```


OpenGL API

- glEnable/glDisable(GL_STENCIL_TEST);
- glStencilFunc(function, reference, mask);
- glStencilOp(stencil-fail,
  stencil-pass_depth-fail,
  stencil-pass_depth-pass);
- glStencilMask(mask);
- glClear(… | GL_STENCIL_BUFFER_BIT);

Request a Stencil Buffer

- If using stencil, request sufficient bits of stencil
- Implementations may support from zero to 32 bits of stencil
- 8, 4, or 1 bit are common possibilities
- Easy for GLUT programs:

```
glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB |
                    GLUT_DEPTH | GLUT_STENCIL);
glutCreateWindow("stencil example");
```

Stencil Test

- Compares reference value to pixel's stencil buffer value
- Same comparison functions as depth test, for glStencilFunc(function, reference, mask);
  - NEVER, ALWAYS
  - LESS, LEQUAL
  - GREATER, GEQUAL
  - EQUAL, NOTEQUAL
- Bit mask controls comparison
  ((ref & mask) op (svalue & mask))
- svalue = stencil buffer value

Stencil Operations

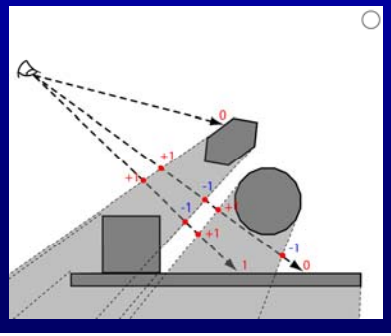
- Stencil side effects of
  - Stencil test fails
  - Stencil test passes and Depth test fails
  - Stencil test passes and Depth test passes
- Possible operations for glStencilOp(stencil_fail, stencil-pass_depth-fail, stencil-pass_depth-pass)
  - Increment, Decrement
  - Keep, Replace
  - Zero, Invert
- Way stencil buffer values are controlled

Stencil Write Mask

- Bit mask for controlling write back of stencil value to the stencil buffer
- Applies to the clear too!
- Stencil compare & write masks allow stencil values to be treated as sub-fields

Shadow Volumes

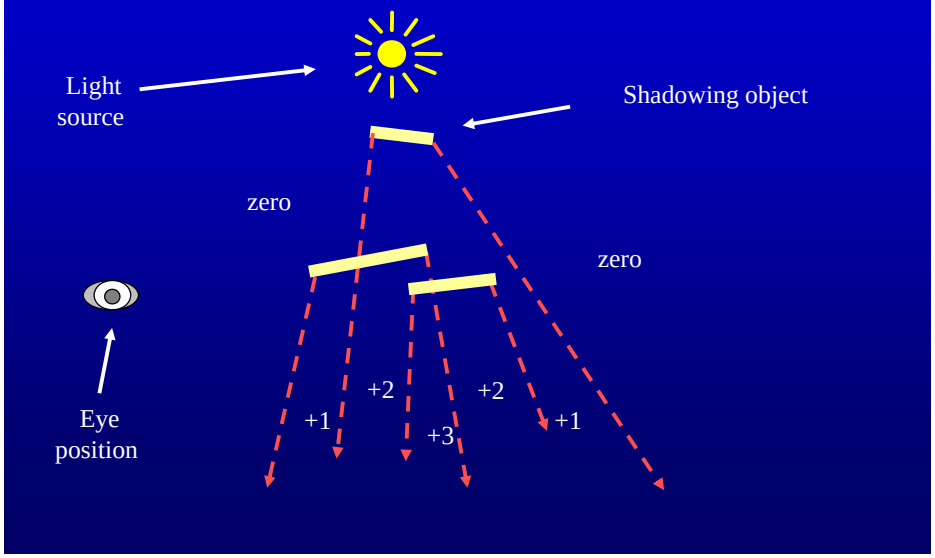
- Principe
  - pour chaque *shadow "casters"*
    - construire un volume d'ombre
  - pour chaque fragment dessiné
    - compter combien de fois on entre/sort d'un volume
      - > 0 : dans l'ombre
      - = 0 : dans la lumière



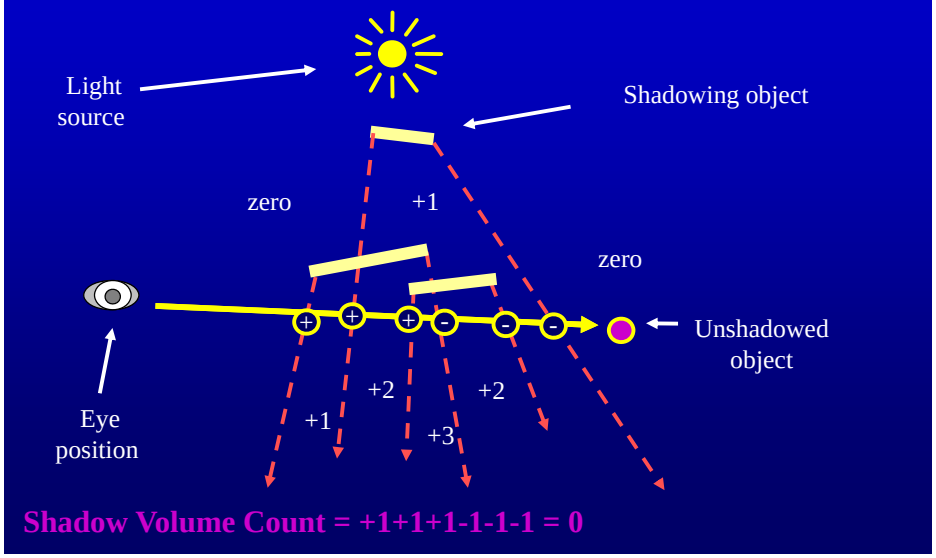
Counting Shadow Volume Enter/Leaves With a Stencil Buffer (*Zpass* approach)

- Render scene to initialize depth buffer
  - Depth values indicate the closest visible fragments
- Use a stencil enter/leave counting approach
  - Draw shadow volume twice using face culling
    - 1st pass: render *front* faces and *increment* when depth test passes
    - 2nd pass: render *back* faces and *decrement* when depth test passes
  - Don't update depth or color
- Afterward, pixel's stencil is non-zero if pixel in shadow, and zero if illuminated

Why Eye-to-Object Stencil Enter/Leave Counting Approach Works

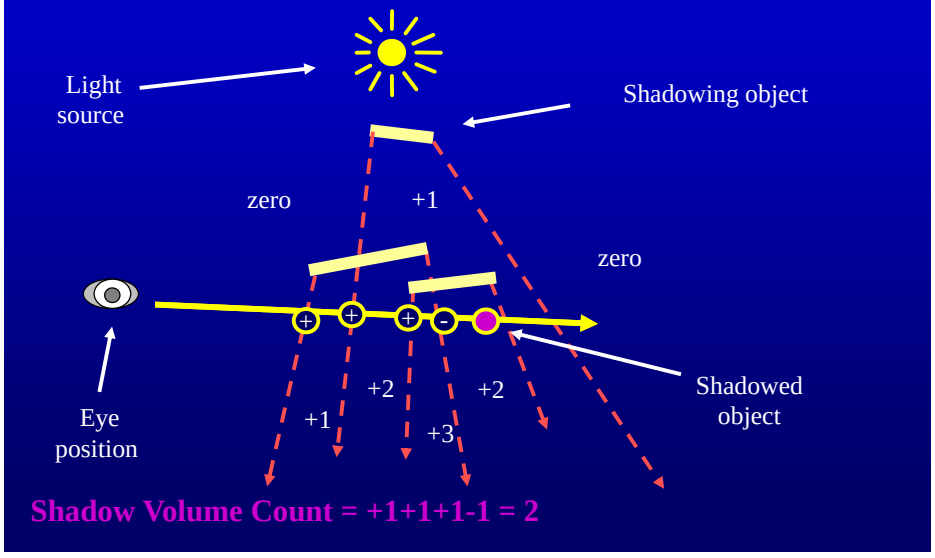


Illuminated, Behind Shadow Volumes



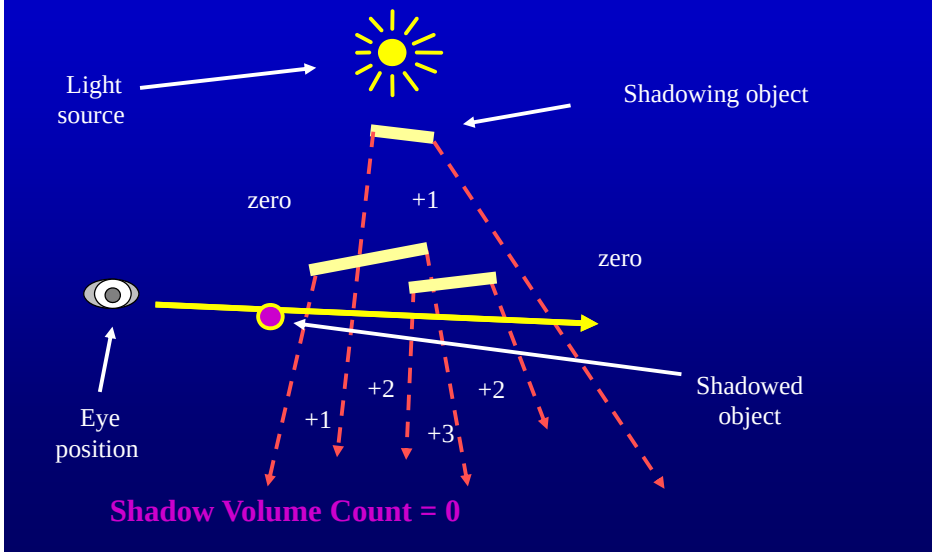
Shadow Volume Count = +1+1+1-1-1-1 = 0

Shadowed, Nested in Shadow Volumes



Shadow Volume Count = +1+1+1-1 = 2

Illuminated, In Front of Shadow Volumes



Shadow Volume Count = 0

Shadow Volumes (2/3)

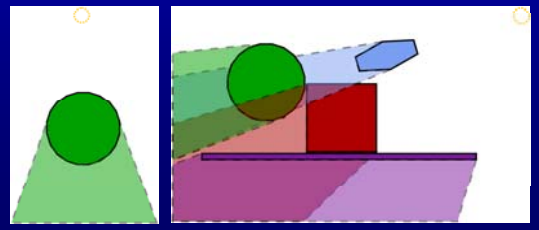
- Comment?
  - construire les volumes d'ombres
    - ⊙ trouver la silhouette des objets vus depuis la source
    - ⊙ construire des *quads* infinis s'appuyant
      - – sur la source
      - – sur chaque arête de silhouette
  - compter les entrées/sorties
    - ⊙ utiliser le *stencil buffer*

Trouver la silhouette

- Algorithme
  - pour chaque arête du modèle :
    - ⊙ identifier les faces gauche/droite et leurs normales
    - ⊙ calculer les prod. scal. normales/vecteur vers la source
    - ⊙ marquer comme silhouette si de signe différents
  - fait sur le CPU, possibilité sur le GPU
- Requiert les infos d'adjacence du maillage
- Calcule un sur-ensemble de la silhouette

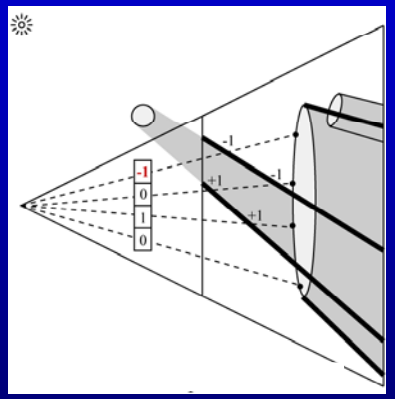
Construire les volumes d'ombres

- On extrude les arêtes de silhouette vers l'∞
- On obtient des *shadow quads*
  - ces *shadow quads* sont orientés
    - ⊙ *front* ou *back facing*
  - ils forment des volumes d'ombres imbriqués
- Dans l'ombre = dans au moins un volume



Stenciled Shadow Volumes (1/3)

- Premier rendu de la scène
  - Initialise le *Z-buffer*
- Rendu du volume d'ombre
  - Pour chaque *front facing shad. quad*
    `glStencilOp(GL_KEEP,GL_KEEP,GL_INCR);`
  - Pour chaque *back facing shad. quad*
    `glStencilOp(GL_KEEP,GL_KEEP,GL_DECR);`
- Deuxième rendu de la scène
  - Pour la partie éclairée
    `glStencilFunc(GL_EQUAL,0,~0);`



Z pass

Code : premier rendu *de la scène sans ombre et que l'ambiant*

```
glEnable(GL_LIGHTING);
glEnable(GL_LIGHT0); // enable light source
glEnable(GL_DEPTH_TEST); // standard depth testing
glDepthFunc(GL_LEQUAL);
glDepthMask(1);
glDisable(GL_STENCIL_TEST); // no stencil testing (this pass)
glColorMask(1,1,1,1); // update color buffer
glClearStencil(0); // clear stencil to zero
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT |
    GL_STENCIL_BUFFER_BIT);
renderScene(); \\  Ne calculer que l'ambiant
```


Code : rendu *shadow volumes*

```
glDepthMask(0);           // Ne pas écrire ds Z-buffer
glStencilFunc(GL_ALWAYS, 0, ~0);
glEnable(GL_STENCIL_TEST);
glEnable(GL_CULL_FACE);
glColorMask(0,0,0,0);     // pas modifier framebuffer
// front-facing quads
glCullFace(GL_BACK);
glStencilOp(GL_KEEP, GL_KEEP, GL_INCR);
draw_shadow_quads();
// back-facing quads
glCullFace(GL_FRONT);
glStencilOp(GL_KEEP, GL_KEEP, GL_DECR);
draw_shadow_quads();
glColorMask(1,1,1,1);
glDepthMask(1);         // On peut écrire ds Z-buffer
```


Code : deuxième rendu de la scène

```
// La scène est rendue à nouveau avec le test de
   stencil buffer permettant de mettre à jours
   uniquement les pixels étiquetés par le « Shadow
   Volume »

glStencilFunc(GL_EQUAL, 0, ~0);
glStencilOp(GL_KEEP, GL_KEEP, GL_KEEP);
glEnable(GL_STENCIL_TEST);
glDepthFunc(GL_EQUAL);
glEnable(GL_LIGHTING);
renderScene();
```


Extend the Shadow Volume

- **Given a Light Source, Extend a Shadow Volume from a Polygon**
- The following routine extendVolume renders with OpenGL a loop of quads extended from an n-side polygon with co-planar vertices v away from a light.
- In the case of a local (positional) light, localLight should be non-zero, and then the lightPosition is considered a 3D position.
- In the case of an infinite (directional) light, localLight should be zero, and then lightPosition is treated as a 3D direction.
- The extendDistance parameter should be a sufficiently large positive value that ensures that the extended vertices are always extended beyond the view frustum.
- The polygon itself is *not rendered by this routine.*

Extend the Shadow Volume

```
typedef GLfloat POSITION[3];
void extendVolume(int n, POSITION v[ ],
                  int localLight, POSITION lightPosition,
                  float extendDistance)
{
    POSITION extendedVertex, extendDirection;
    int i;
    // If light source is infinite (directional)...
    if (!localLight) {
        // compute direction opposite from light source direction.
        extendDirection[0] = -lightPosition[0];
        extendDirection[1] = -lightPosition[1];
        extendDirection[2] = -lightPosition[2];
    }
```


Extend the Shadow Volume

```
glBegin(GL_QUAD_STRIP);
// If light source is local (positional)...
if (localLight) {
    for (i=0; i<n; i++) {
        glVertex3fv(v[i]);
        // Compute per-vertex direction from vertex away from the light
        source.
        extendDirection[0] = v[i][0] - lightPosition[0];
        extendDirection[1] = v[i][1] - lightPosition[1];
        extendDirection[2] = v[i][2] - lightPosition[2];
        // Compute extended vertex.
        extendedVertex[0] = v[i][0] + extendDirection[0] * extendDistance;
        extendedVertex[1] = v[i][1] + extendDirection[1] * extendDistance;
        extendedVertex[2] = v[i][2] + extendDirection[2] * extendDistance;
        glVertex3fv(extendedVertex);
    }
```


Extend the Shadow Volume

```
    // Repeat initial 2 vertices to close the quad strip loop.
    glVertex3fv(v[0]);
    extendDirection[0] = v[0][0] - lightPosition[0];
    extendDirection[1] = v[0][1] - lightPosition[1];
    extendDirection[2] = v[0][2] - lightPosition[2];
    extendedVertex[0] = v[0][0] + extendDirection[0] * extendDistance;
    extendedVertex[1] = v[0][1] + extendDirection[1] * extendDistance;
    extendedVertex[2] = v[0][2] + extendDirection[2] * extendDistance;
    glVertex3fv(extendedVertex);
// otherwise, light source is infinite (directional)...
```


Extend the Shadow Volume

```
    } else {
        for (i=0; i<n; i++) {
          glVertex3fv(v[i]);
          // Compute extended vertex.
          extendedVertex[0] = v[i][0] + extendDirection[0] * extendDistance;
          extendedVertex[1] = v[i][1] + extendDirection[1] * extendDistance;
          extendedVertex[2] = v[i][2] + extendDirection[2] * extendDistance;
          glVertex3fv(extendedVertex);
          }
          // Repeat initial 2 vertices to close the quad strip loop.
          glVertex3fv(v[0]);
          extendedVertex[0] = v[0][0] + extendDirection[0] * extendDistance;
          extendedVertex[1] = v[0][1] + extendDirection[1] * extendDistance;
          extendedVertex[2] = v[0][2] + extendDirection[2] * extendDistance;
          glVertex3fv(extendedVertex);
      }
  glEnd()
  }
```

Extend the Shadow Volume

Here is an example using extendVolume to render the shadow volume for a triangle:

```
POSITION triangle[3] = { { 0, 0, 0 }, { 1, 0, 0}, { 0, 1, 0 } };
POSITION lightPosition = { 1, 1, 7 };

glDisable(GL_LIGHTING);
extendVolume(3, triangle,
        1, lightPosition,    // local light
        100000.0);           // big positive number
glEnable(GL_LIGHTING);
```

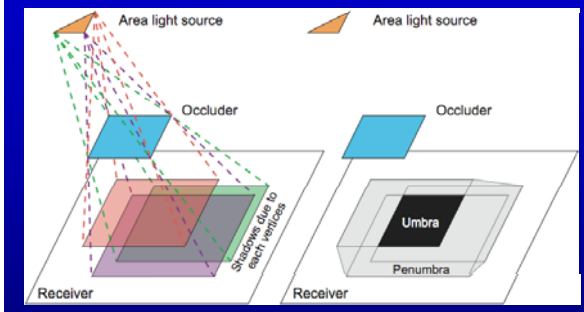

Shadow volumes: bilan

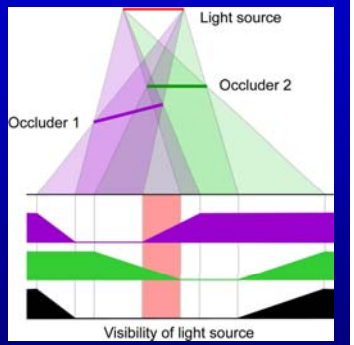
- Avantages :
  - ombres précises
  - positions quelconques source/caméra
  - robuste si bien programmé,
- Inconvénients :
  - calcul de la silhouette (sur CPU, év. long)
  - scènes bien modélisées préférables
  - deux rendus de la scène + rendu des volumes

Ombres douces

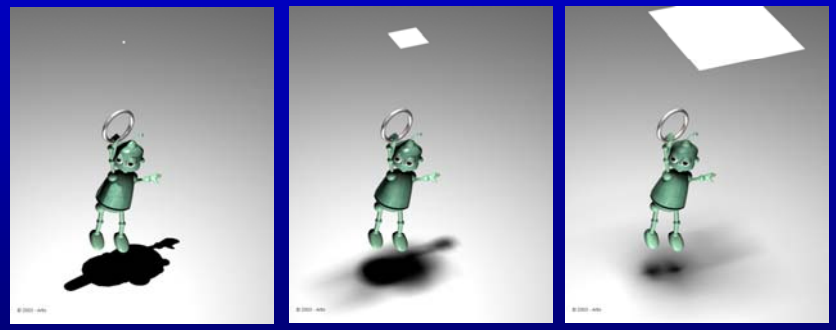
- Algorithmiquement plus compliqué
  - problème de visibilité point-surface
    - au lieu de point-point
    - silhouette ?
  - ombre de la somme ≠ somme des ombres
- Plusieurs algorithmes approximatifs
  - suffisants pour l'oeil humain
  - voir survol par Hasenfratz et al.

Ombres et pénombre





Problèmes de silhouette



Ombres douces par *sampling*

- Accumulation d'ombres :
  - calculer plusieurs ombres ponctuelles
  - additionner et moyenner les résultats
  - *accumulation buffer*
  - nombre d'échantillons élevés
    - temps de calcul multiplié par # échantillons

4 échantillons

1024 échantillons

Ombres douces (1/2)

- *Shadow volume* normal
- Pour chaque arête de silhouette :
  - calculer volume englobant la pénombre
  - pour chaque pixel dans ce volume
    - calculer coefficient d'atténuation
- Beau, réaliste mais *fill-rate* multiplié par 2

Penumbra wedges [Sig03]
U. Assarson, T. Möller

Ombres douces (2/2)

Rendering Fake Soft Shadows with Smoothies[SoR03]
E. Chan, F. Durand

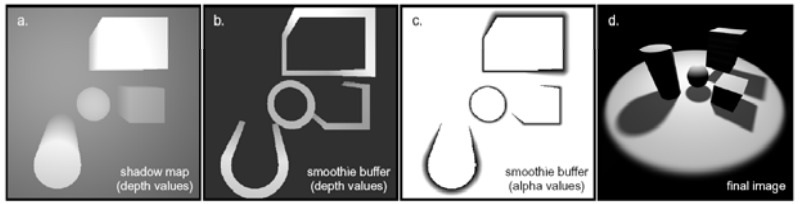


Figure 1: *Smoothie algorithm overview. (a) We first render a shadow map from the light's viewpoint. Next, we construct geometric primitives that we call* smoothies *at the objects' silhouettes. (b) We render the smoothies' depth values into the smoothie buffer. (c) For each pixel in the smoothie buffer, we also store an alpha value that depends on the ratio of distances between the light source, blocker, and receiver. (d) Finally, we render the scene from the observer's viewpoint. We perform depth comparisons against the values stored in both the shadow map and the smoothie buffer, then filter the results. The smoothies produce soft shadow edges that resemble penumbrae.*

Penumbra maps: Approximate soft shadows in Real-time [SoR03]
E. Chan, F. Durand

Résumé & conclusion

- Shadow maps :
  - Stable, robuste, facile, rapide, aliasage
- Shadow volumes :
  - Beau, difficile, complexité algorithmique
- Ombres douces
  - Complexe, lent, beau

- Beaucoup de papiers récents
  - Bibliographie en ligne

    `http://artis.imag.fr/~Xavier.Decoret/index.fr.html`