Eurographics Workshop on Intelligent Cinematography and Editing (2016)
M. Christie, Q. Galvane, A. Jhala, and R. Ronfard (Editors)

Contact Visualization

J-E. Marvie[1], G. Sourimant[1] and A. Dufay[1]

[1]Technicolor

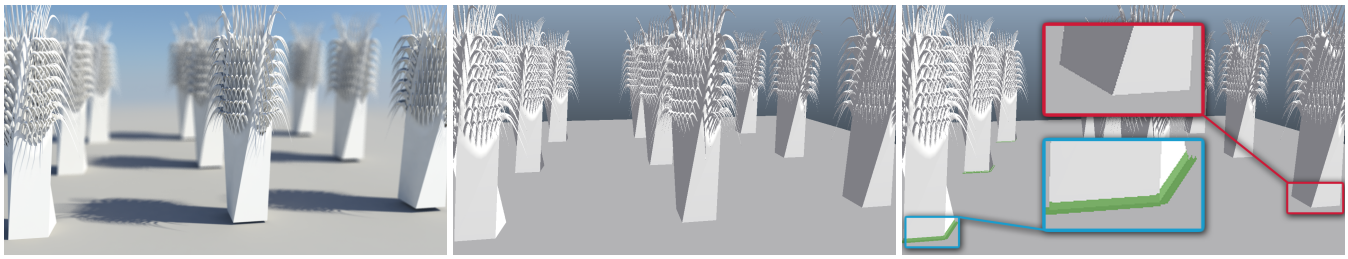


Figure 1: During assets initial setup, misalignments may lead to floating objects or inter-penetrations at the end of the pipeline (Left), leading to rollbacks on the modeling or animation steps. The lack of shading and shadowing prevents from detecting such errors easily (Middle). We propose an integrated real-time solution that provides a visual feedback to determine whether 3D objects are in contact with each other or not (Right). In the blue rectangle, green pixels overlaid by our technique indicate that objects are in contact as expected; in the red rectangle, no contact is detected while there should be one.

Abstract

We present in this paper a production-oriented technique designed to visualize contact in real-time between 3D *objects. The motivation of this work is to provide integrated tools in the production workflow that help artists setting-up scenes and assets without undesired floating objects or inter-penetrations. Such issues can occur easily and remain unnoticed until shading and/or lighting stages are set-up, leading to retakes of the modeling or animation stages. With our solution, artists can visualize in real-time contact between* 3D *objects while setting-up their assets, thus correcting earlier such misalignments. Being based on a cheap post-processing shader, our solution can be used even on low-end* GPUs*.*

Categories and Subject Descriptors (according to ACM CCS): Computer Graphics [I.3.3]: Three-Dimensional Graphics and Realism—Display Algorithms Computer Graphics [I.3.7]: Three-Dimensional Graphics and Realism—Animation;

1. Introduction

In film production pipelines implying CGI, modeling and animation are usually the first steps to be performed. Animations of 3D models are thus generated without lighting and shadowing information. However, these information can be crucial when it comes to positioning 3D objects within the scene. For instance, in case of a walking virtual 3D character, its feet shall touch the ground to produce realistic images. However, without shadows visualization, this simple constraint can become a very tedious task (Figures 1, 4). This problem may only appear at the end of the production process, when lights are set up in the 3D scene, implying costly retakes, back to the animation step. Similarly, undesired inter-penetrations may also remain unnoticed until shading is performed (Figure 2).

In this paper, we propose a solution to overcome this problem through the use of a simple image filter that detects contact and inter-penetration between 3D objects. When a contact is detected, or when objects are very close to each other, the color of the corresponding pixels is altered to provide a visual feedback to the artists (Figure 1). This filter is cheap and has been implemented on the GPU as a post-processing shader. This approach fits well in production workflows as only errors visible from the camera need to be corrected, contrary to games or virtual worlds where contacts need to be consistent everywhere.

After discussing existing techniques on contact detection and visualization between 3D objects, we describe the core of our post-processing algorithm. Its performances and limitations are then discussed before concluding the paper.

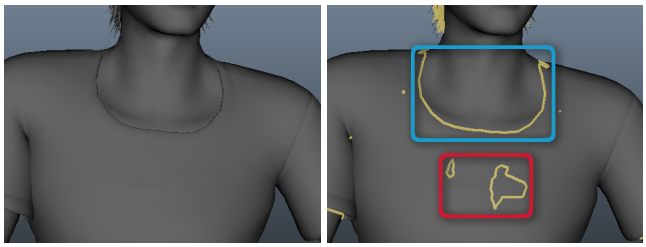


Figure 2: Inter-penetration detection example. Left: a Lambert-shaded character during animation setup. Right: our solution enlightens both that the shirt is in contact with the neck as expected (blue rectangle), and that undesired inter-penetration between the shirt and the torso occur (red rectangle).

(a) Without shadows (b) With shadows

Figure 3: Contacts become clearly visible with shadows, but they are too complex to compute in some cases, for instance when animations are set up.

2. Related Works

As already stated in the introduction, since contact or inter-penetration mistakes imply errors in the lighting passes, shadowing techniques [ESAW11] can be used to determine visually whether two 3D objects are in contact (Figure 3). However, since these methods are generally too costly to be computed in real-time at modeling or animation steps, and since they are completely different from the technique described in this paper, we will not detail them in this document.

Our solution can be more closely related to collision detection techniques in physics simulations [JTT00, KHI*07]. One of the main problems to be solved in physics simulation is the detection of collision and / or inter-penetration of different 3D objects. Traditionally, collision detection is performed in object space (*i.e.* in the 3D geometric space) on a per-object basis, while our technique enlightens contacts in image-space on a per-pixel basis.

The goal of collision detection in object-space is to determine physical forces applied to the 3D objects within the scene [JTT00]. It thus requires precise geometric information on the contacts themselves to evaluate the forces resulting from these contacts. On the other hand, our method only detects the existence of contacts and does not need precise description of the contacts themselves. As a consequence it can be computed much faster and without using object-based computations.

There also exist in the literature image-based collision detection techniques [BWS98, MOK95, FAFB08]. However, as in the object-based case, the goal is to compute a physics simulation and thus determine precisely forces that are applied to the 3D objects. These methods thus require more complex computations, such as:

- Pre-computation of objects intersection using bounding-boxes
- Additional renderings from orthogonal viewpoints
- Multiple renderings in layered depth images

On the other hand, our contact detection method does not require additional renderings. It only requires to store a 3D position and an object identifier per pixel at render time, with thus almost no overhead.

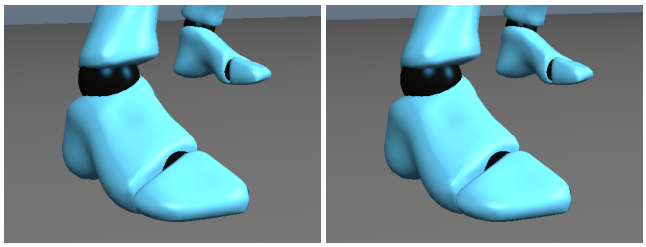


Figure 4: Left: the feet of the character are above the ground. Right: they are in contact with it. As one can see, it is impossible to tell from this point of view whether the feet are in contact with the ground or not, without additional visual feedback.

3. Contact Visualization

3.1. Principle

Our contact visualization filtering solution is built upon the deferred pipeline principle, where additional information is stored in a texture at render time, on a per-pixel basis. In our solution, we store (or use†) two pieces of information associated to the object that is rasterized: its geometric 3D position $\mathbf{P}_{pos}$ (expressed in scene units) and a unique object identifier $\mathbf{P}_{id}$. In a subsequent rendering pass, the complete texture (shaded scene and additional information) is mapped on a full-screen quad and the filtering process is performed in an associated fragment shader.

During the filtering stage, for a given pixel $\mathbf{P}$ and its associated data $\mathbf{P}_{pos}$ and $\mathbf{P}_{id}$, a contact is detected if there exist in its neighborhood at least one pixel $\mathbf{P}'$ for which:

$$\begin{cases} \|\mathbf{P}_{pos} - \mathbf{P}'_{pos}\| & < & \varepsilon_{pos} \\ \mathbf{P}_{id} & \neq & \mathbf{P}'_{id} \end{cases} \quad (1)$$

The value of the scalar ε_{pos} depends on the scene and reflects the artist-driven tolerance below which two objects can be considered to be in contact. In practice, once a contact is detected for a pixel, the visual feedback can take different forms. For instance, the input pixel color can be blended with an artist-defined color in case of contact and kept unaltered otherwise. The amount of alteration of the input pixel could also be proportional to the portion of contacting pixels in the neighborhood of the input pixel.

† The geometric position can be inferred from the internal depth buffer used for z-buffering rather than stored explicitly, thus reducing the memory footprint of our solution.

We propose in this paper two different algorithms to detect contacts. They differ mostly from the sampling space they use, and the algorithm choice is driven by a trade-off between performances and ease-of-use.

Image-space algorithm. On the one hand, one can consider standard filtering by sampling the input texture directly to detect contacts (Algorithm 1), by looking at neighboring pixels in a fixed window search size. The main advantage of this approach is that regardless of the scene content, the number of samples per pixel fetched from the input texture remains constant, and the texture caching behavior is predictable. This leads to high performances even on very low-end graphics hardware, but may be less user-friendly than the alternative camera-space algorithm.

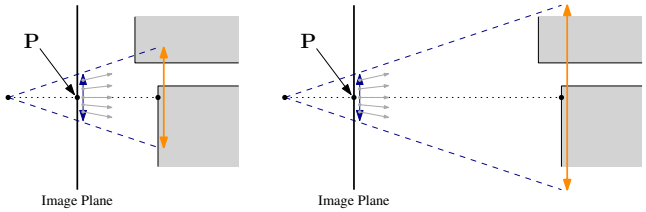
Algorithm 1 Contact detection in image space

Input: Textures $\mathbf{T}^*$ storing colors, positions *pos* and identifiers *id*.
Input: Parameter $\mathbf{C}_{col}$, the contact color
Input: Parameter ε_{pos}, the distance threshold (in scene units)
Input: Parameter κ_1, the filtering kernel size (in pixels)
for all texels $\mathbf{P}$ **do**
  Read from $\mathbf{T}^*$ the pixel color $\mathbf{P}_{col}$
  Read from $\mathbf{T}^*$ the pixel data $\mathbf{P}_{pos}$ and $\mathbf{P}_{id}$
  Initialize the blend factor: $\alpha = 0$
  for all samples i within κ_1 **do**
    Compute the sample texture coordinates
    Read from $\mathbf{T}^*$ the sample data $\mathbf{P}'_{pos}$ and $\mathbf{P}'_{id}$
    if $\mathbf{P}_{id} \neq \mathbf{P}'_{id}$ && $\|\mathbf{P}_{pos} - \mathbf{P}'_{pos}\| < \varepsilon_{pos}$ **then**
      Modify the value of α
    end if
  end for
  $\mathbf{P}_{col} = \alpha \cdot \mathbf{C}_{col} + (1-\alpha) \cdot \mathbf{P}_{col}$
end for

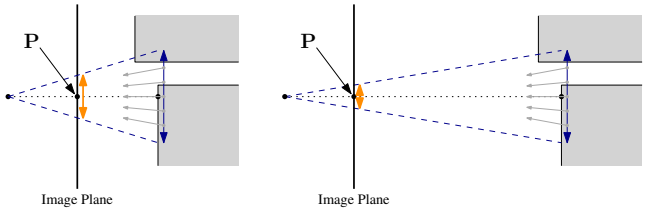
Camera-space algorithm. On the other hand, filtering samples can be defined first in camera-space as 3D offsets to $\mathbf{P}_{pos}$ read from the input texture(s), that are then projected back in image space to read corresponding coordinates $\mathbf{P}'_{pos}$ and identifier $\mathbf{P}'_{id}$ (Algorithm 2). With this method, many different objects far away from the camera will generate almost no contact visualization. This is consistent with the initial idea where contact visualization helps artists set up correctly their assets to avoid floating objects or inter-penetrations: there may be no need to correct a scene with floating objects when the impact of this error is less than a pixel. However, in case of important close-ups, this method may lead to texture fetches far away from the central pixel position, and thus break the texture cache consistency at runtime. This is because the image size of the filtering kernel in this case depends on the pixels' 3D coordinates, contrary to the image-space filtering method. On the pragmatical side, current graphics hardware generally have large amounts of texture caches, so this issue may only occur either on much older hardware, or with improbable shader parameterization (*e.g.* using a kernel that covers the whole image). The kernel of this filter being designed to fit the underlying geometry, it produces more accurate results and is more artist-friendly in term of parameterization. The difference in behavior between both algorithms is illustrated on Figure 5.

Algorithm 2 Contact detection in camera space

Input: Textures $\mathbf{T}^*$ storing colors, positions *pos* and identifiers *id*.
Input: Parameter $\mathbf{C}_{col}$, the contact color
Input: Parameter ε_{pos}, the distance threshold (in scene units)
Input: Parameter κ_2, the filtering kernel size (in scene units)
for all texels $\mathbf{P}$ **do**
  Read from $\mathbf{T}^*$ the pixel color $\mathbf{P}_{col}$
  Read from $\mathbf{T}^*$ the pixel data $\mathbf{P}_{pos}$ and $\mathbf{P}_{id}$
  Initialize the blend factor: $\alpha = 0$
  for all samples i **do**
    Compute sample position $\mathbf{S}_{pos}$ in camera-space as a 3D offset of $\mathbf{P}_{pos}$ within the kernel limits κ_2
    Project $\mathbf{S}_{pos}$ to get the associated texture coordinates
    Read from $\mathbf{T}^*$ the sample data $\mathbf{P}'_{pos}$ and $\mathbf{P}'_{id}$
    if $\mathbf{P}_{id} \neq \mathbf{P}'_{id}$ && $\|\mathbf{P}_{pos} - \mathbf{P}'_{pos}\| < \varepsilon_{pos}$ **then**
      Modify the value of α
    end if
  end for
  $\mathbf{P}_{col} = \alpha \cdot \mathbf{C}_{col} + (1-\alpha) \cdot \mathbf{P}_{col}$
end for



(a) Image-space sampling. Here the sampling space is constant in image size (blue arrow), and samples in object space can be very far away from the considered 3D point (orange arrow), thus leading to inconsistent local geometry inspection.



(b) Camera-space sampling. In that case the size of the sampling search varies in image space (orange arrow), while it remains constant in object space (blue arrow), leading to a more more accurate and artist-friendly filter parameterization.

Figure 5: Behavior difference of the image and camera-based algorithms. Left column: case of objects close to the camera. Right column: case of objects far from the camera.

3.2. Implementation

During the initial render pass, objects need to store for each pixel their identifier and position in a dedicated render target. Though 16 bits-per-component (bpc) may be sufficient in general to measure small geometric distances, it limits the number of object identifiers to 65536. In production scenes comprising a huge number of objects, this could be too small. We thus recommend using 32 bpc

render targets. As for geometric positions, they can be stored either explicitly in the same render target using three floating point values, or they can be inferred in the post-processing filtering pass using the internal depth buffer used for z-buffering and the camera intrinsic parameters.

In the final rendering pass, where contacts are searched for (Equation 1), any sampling strategy should work. However, for performance reasons, we use Poisson-disk samples (defined as offsets from the central point in the unit disk) rather than say box filtering, so as to keep a constant number of filtering samples regardless of the filter kernel size (which varies with algorithm and user parameters). Increasing the kernel size will thus have a much smaller impact on performance, which may decrease slightly due to texture cache misses rather than because of an increased number of texture fetches.

Listing 1 depicts a simple GLSL snippet in which the input pixel color is blended with an artist-driven color as soon as a contact is detected in the vicinity of said pixel. In this example, the kernel size is defined in image space rather than camera space for the simplicity of the exposition.

4. Results and discussion

We integrated our contact visualization solution in Autodesk's Maya, within an in-house pre-visualization viewport plug-in. All the following tests have been performed on a desktop computer with an NVIDIA GeForce GTX 480. All images are rendered at a 1280×720 definition. Unless otherwise stated, tests have been performed with 32 bpc render targets.

4.1. Performance considerations

First, we measured the overhead of such contact shader. Figure 6 illustrates the rendering time spent (in milliseconds) to render the images illustrated in Figure 1 (middle and right). The same image was rendered during approximately 30 seconds to provide a fair amount of sample measures. The three curves correspond to a shader without contact detection (blue), with detection in image-space (green) and in camera-space (red). For this image, one can see that the overhead is similar for both sampling spaces, and approximately equal to 1 millisecond. This example represents a typical image, with some background pixels that are not processed (we do not compute irrelevant contacts on background pixels). In the worst case, when all pixels are processed, the overhead costs approximately 1.2 milliseconds. The choice of the sampling space seems to have a negligible impact on performances.

We also tested the performance impact of the kernel size in the case of image-based sampling. Figure 7 illustrates the median rendering time (over 30 seconds) for different kernel radii in pixels. We tested values 0 (no shader), {1..32}, 64, 128 and 256 pixels. Surprisingly, on this kind of hardware, fetching texels far away from the central one does not seem to impact performances that much.

Finally, we measured the impact of using 16 bpc render targets instead of 32 bpc (Figure 8). With such buffer precision, the number of objects that can be discriminated through their object id is decreased. It should thus be used we scenes with low to medium

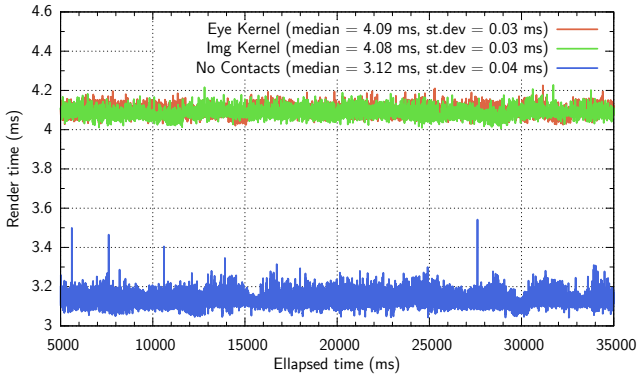


Figure 6: Scene render time with contacts computed in image space, in camera space, or without contacts.

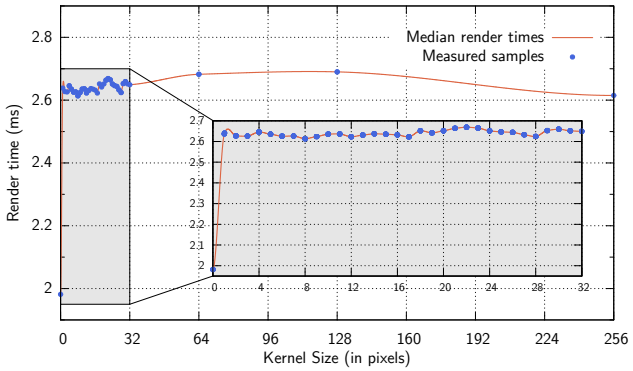


Figure 7: Median render time *vs.* kernel size. The rectangular zone in the center of the graph is a zoom on the first 32 samples.

objects number. Unsurprisingly, the whole rendering time is decreased while using 16 bpc render targets. It is also worth noting that the contacts shader overhead is smaller when reading 16 bits floating values rather than 32 (approximately 1.3 times faster in this particular example).

4.2. Limitations

This contact and inter-penetration previsualization solution suffers from several limitations, which are inherited from the limitations of any deferred shading solution.

First, contacts or inter-penetrations cannot be visualized on transparent objects. As a matter of fact, such objects do not write in the depth buffer and shall not write other pixel data than their own color. In standard deferred pipelines, transparent objects are generally drawn after the deferred process has been applied to opaque objects.

Second, this technique can miss contact detections for some pixels both because of the depth map representation of the scene and the sampling process. Notice however that this technique does not produces false positives (*i.e.* it does not detect contacts for pixels where there isn't one).

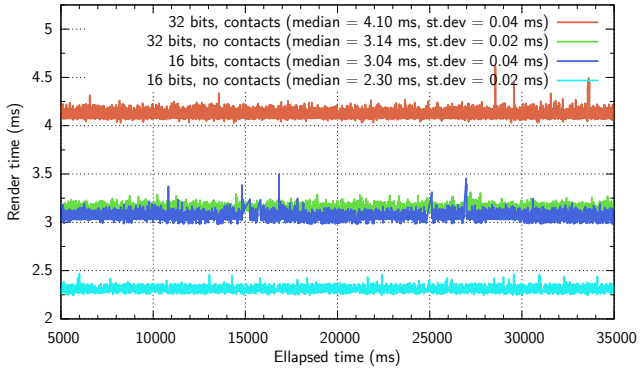


Figure 8: Render time *vs.* render targets bit depth, with and without contact detection.

Some deferred pipeline techniques based on the depth of the pixels take advantage of depth-peeling to remove missing information in the initial depth layer (see for instance [BS09]). In our case, a contact would be searched for in successive depth layers with increasing depth order. It is worth noting that some previsualization solutions already implement depth-peeling for high quality transparency rendering, and could thus directly benefit from such enhanced depth information to compute contacts more accurately. However, such peeling solutions have a strong impact on graphics performances and may not fit the constraints of animation workflows, due to the allocation and read-back of many floating-point frame buffers.
Some contacts can be missed in case of stochastic sampling of the pixel's neighborhood. Increasing the number of samples leads to a decrease of the contact miss rate, at a computation cost roughly linear with the number of samples. However, this is generally not an issue, as zones with missed contacts appear as dithered strokes instead of filled ones.

Finally, our solution relies on per-object identifiers. If different meshes are merged into a single one, only topological information on triangle- or quad-level can be inferred. For instance, we can detect in the neighborhood of a pixel if two triangles intersect without sharing an edge. It would require however more advanced graphics capabilities, as topology analysis can only be done in geometry or tessellation shaders, if one expects real-time feedback.

5. Conclusion

The simple technique presented in this paper allows for real-time visualization of contact and inter-penetration between 3D objects in CGI images. Thanks to this post-processing approach, there is no need for inter-objects collision detection in geometric space. As such, the computational cost of our method is independent of the complexity of the scene geometry.

Our technique was successfully embedded within Autodesk's Maya viewport. As such, artists can build directly 3D assets and scenes while avoiding undesired lack of contact or inter-penetration, without any additional costly tool or rendering effect (*e.g.* shadows, ambient occlusion, etc.). This initial quality assessment reduces dramatically the number or retakes due for instance to erroneous lighting, leading to a fair amount of time gain on the whole production workflow.

References

[BS09] BAVOIL L., SAINZ M.: Multi-layer dual-resolution screen-space ambient occlusion. In *SIGGRAPH 2009: Talks* (New York, NY, USA, 2009), SIGGRAPH '09, ACM, pp. 45:1–45:1. 5

[BWS98] BACIU G., WONG W. S.-K., SUN H.: Recode: an image-based collision detection algorithm. In *Computer Graphics and Applications, 1998. Pacific Graphics '98. Sixth Pacific Conference on* (1998), pp. 125–133. 2

[ESAW11] EISEMANN E., SCHWARZ M., ASSARSSON U., WIMMER M.: *Real-Time Shadows*. A.K. Peters, 2011. 2

[FAFB08] FAURE F., ALLARD J., FALIPOU F., BARBIER S.: Image-based Collision Detection and Response between Arbitrary Volumetric Objects. In *ACM SIGGRAPH/Eurographics Symposium on Computer Animation* (Dublin, Irlande, July 2008), pp. 155–162. 2

[JTT00] JIMÉNEZ P., THOMAS F., TORRAS C.: 3d collision detection: A survey. *Computers and Graphics 25* (2000), 269–285. 2

[KHI*07] KOCKARA S., HALIC T., IQBAL K., BAYRAK C., ROWE R.: Collision detection: A survey. In *Systems, Man and Cybernetics, 2007. ISIC. IEEE International Conference on* (2007). 2

[MOK95] MYSZKOWSKI K., OKUNEV O., KUNII T.: Fast collision detection between complex solids using rasterizing graphics hardware. *The Visual Computer 11*, 9 (1995), 497–511. 2

Appendix

```
// Input MRT sampler (color + pos & id)
uniform sampler2D inTex[2];

// Contact-related uniforms
uniform float ctMaxDist;
uniform float ctKerSize;
uniform vec3  ctColor;
uniform float ctIntensity;

// Nb Poisson-disk samples for contacts
const int nSamples = 15;
// Poisson-disk samples
vec2 poisson[15] = vec2[15](
  vec2(-0.699005, -0.495301), [...]
  vec2( 0.965083,  0.147416)
);

void displayContacts( inout vec4 inColor )
{
  //--------------------------------------//
  // Retrieve the fragment's Id and depth,
  // and compare to the neighborhood
  vec2 tcRef = gl_FragCoord.xy - vec2(0.5);
  vec4 ref = texelFetch( inTex[1], ivec2(tcRef), 0 );
  float mixCoeff = 0.0;

  for( int i = 0; i < nSamples; i++ )
  {
    vec4 sample = texelFetch( inTex[1],
      ivec2( ctKerSize * poisson[i] + tcRef ), 0 );

    // If sample Id is different from ref. Id
    // and they are close enough
    if( ref.w != sample.w &&
      length( sample.xyz - ref.xyz ) < ctMaxDist )

      mixCoeff = ctIntensity;
  }

  inColor.xyz = mix( inColor.xyz, ctColor, mixCoeff );
}

void main()
{
  // Read input color, and eventually modify it
}
```

Listing 1: Minimal GLSL snippet used to detect contacts in image space, with 3D positions explicitly stored.