Noname manuscript No.
(will be inserted by the editor)

Image mapping system for simulating ceramic environments

**Inmaculada Remolar · Miguel Chover ·
Cristina Rebollo · Cristina Gasch**

Received: date / Accepted: date

Abstract Minimizing costs and increasing sales are a goal for every business nowadays. This fact, together with the development of new technologies, have driven the emergence of virtual applications where the customers can configure the product they are interested in only interacting with the images where the products appear. Many applications are available on Internet or app stores for this purpose. In all of them, a high realism is required. However, this fact is directly related to a high cost of storage of data and to the difficulty of generating the images of the scenes where the product is exposed. This paper presents a virtual configurator addressed to tile factories that solves these problems maintaining a high realism. The developed application generates the configurable images by rendering 3D modeled environments and the customization is performed taking advantage of the graphics hardware. It is in charge of performing the tiling of any size tiles in real time. The presented image mapping system is based on the real measurements of the walls or floor of the environment that appear in the image and on the dimensions of the tile to map. Taking these data into account, the application performs the final appearance adapting the final image to the requirements of the user. The presented method reduces the amount of stored information maintaining the realism of the customized images.

Keywords image composition. virtual composing. virtual design. automatic tiling

I. Remolar
Institute of New Imaging Technologies, Universitat Jaume I, 12006 Castellón, Spain
E-mail: remolar@uji.es
Phone number: +34 964 38 77 68
Fax number: +34 964 38 76 78

M. Chover
E-mail: chover@uji.es

C. Rebollo
E-mail: rebollo@uji.es

C. Gasch
E-mail: cgasch@uji.es

1 Introduction

In the business world it is usual that companies try to reduce costs to increase their competitiveness. One important way to achieve this is taking advantage of the technological media. In companies that are dedicated to the manufacture of products, most of them are digitally modeled and rendered in a first stage. These rendered images are shown to the potential customers and only if a product is sold, it is finally manufactured. This is a good method to save money to the companies that avoid accumulating products in their stores. If the product can have various finishes, virtual simulators allow the user to configure it on demand [28]. This kind of applications are available on Internet and on app stores. They make it possible that the user can change the appearance of an object on the image only clicking over it and later, over a texture that can be applied on the product. The object changes its looking in real time, appearing configured with the selected material [12, 18].

Realism is a must in virtual simulators. The image has to show the product completely immersed in the scene and, independently of the finish of the configurable object, this environment has to be as realistic as possible, looking even as a real world picture. Different commercial fields use these applications: furniture industry, painting, clothes, tiles, ... However, all of them requires the same thing, that the objects quickly change their appearance on demand in the represented environment maintaining the realism.

Some of the most popular virtual simulators have been analyzed, regardless of the business for which they are addressed. They have been classified in two main groups. On one hand, applications that offer to the customers the possibility of reproducing their own scenario and seeing it in 3D. On the other hand, applications based on images that allow the user to personalize the scenario selecting parts of the image.

In the first group, applications are basically focused on establishing the disposition of the walls that form a room on a 2D plane [24, 9]. Then, users place the objects offered by the virtual simulators in the scene and configure them according to their preferences and requirements. Usually, the possibility of changing the display mode from a 2D aerial view to a 3D view is offered to the user. However, these virtual simulators generally suffer from lack of realism, because they do not offer a photorealistic rendering of the environment.

The second group in our classification, the virtual simulators based on images, show the realism that do not have the previous one. After analyzing the most popular ones, they generally provide different scenarios to show all the possibilities of the product that is been promoting. In all of them, the object to be configured appears immerse in environments adapted to it. The images that are available at these virtual configurators usually are taken from real photographs of scenarios where the object to configure appears. Moreover, in some virtual simulators, the users are allowed to upload their own photographs in order to change the appearance of some objects that appear in them, usually walls or floors [17, 1].

Within the virtual simulators based on images, some applications take advantage of the technology and offer realistic images that have been previously rendered from 3D modeling applications. Realism is obtained in these cases by adjusting the physical illumination, the shadows or some characteristics of the assigned materials. This solution is one of the most popular nowadays because it offers the possibility of representing environments with different styles without the necessity of recreating them in the reality [21]. In order to customize the object, the images need to be processed to isolate the areas where the product appears. Some analyzed applications generate in a pre-process masking images that perform this function. Other analyzed methods in this group require user interaction to identify the configurable object. The user clicks over the area where the configurable object appears and using image based techniques, the object is isolated [27]. Finally, other applications require that the user draws over the image some lines that will demarcate the area to personalize [5].

In the field of the ceramic, the different sizes of the tiles and the different possible dispositions over the walls or floors have to be considered. The virtual simulators addressed to this product have to control the arrangement of the tiles in addition to the rest of the steps that have to be performed to obtain the required realism. Different methods have been implemented to make this possible, but all of them require that the developed application stores a high amount of images that produces a high cost of data storage.

This paper presents a virtual simulator based on images, designed for the ceramic industry that allows to change the model and disposition of the tiles reducing the storage cost and maintaining a photorealistic finish. This developed simulator is based on images obtained from rendering 3D environments. The realism of these images have been previously achieved managing the modeling software and adjusting the illumination and materials of the objects. Moreover, the presented method is scalable: more tile models can be uploaded to the virtual simulator and the application automatically performs the tiling only taking their dimensions into account. It has been developed taking advantage of the shader programming, so the temporal cost of the performed operations is negligible.

The article is organized as follows. Section 2 reviews the techniques used by the most popular virtual designers based on images. Section 3 analyzes the elements involved in the presented method and the way they are combined to obtain the final image is explained in Section 4. Section 5 presents the evaluation of the method and finally, the conclusions and future work are analyzed in Section 6.

2 Related work

Virtual environments are widely used for e-commerce [16]. As it is said in [3], the possibility of users designing the products on-line exploits the interactivity of the web and enables users to design their own virtual products thus enabling the product development team to learn their own preferences for new products. Virtual configurators represent some environments where the product to sell is immerse. They have been traditionally very popular in fields such as furniture, clothes, painting and tiles, among others This section analyzes the simulators that offer photorealistic images where the user can change the configuration of the product that is on sale.

The virtual configurator presented in this paper is focused on ceramic industry so, after analyzing the methods used by similar applications available on Internet or the digital app markets, the ones addressed to tiles are also studied in this section.

2.1 Virtual simulators based on images

The simulators based on images offer the possibility of choosing scenarios according to different styles: modern, classic, ethnic, ... This assortment allows customers to configure the products they are interested in and see how they look in scenarios that fit better to their preferences.

Focusing the analysis on simulators based on images, two different groups have been classified: based on real photographs and based on rendered 3D scenarios.

Based on real photographs The realism in the images is guaranteed if they are photographs taken from real environments. In order to identify where the customizable object appears in them, some actions have to be performed [10, 23]. Some applications propose to apply a pre-process to the set of images to solve this problem. The most popular solution is to use alpha channel masks that cut off the configurable object in the images. These masks allow the application to extract some slices from the images and, lately, re-build a new one from them [31]. To perform this collage, some images where the object appears configured with all the possible materials have been created. All of them are stored in the database of the applications [22]. The main disadvantage of this solution is the high storage cost. Every area in the image that can be changed needs two alpha channels masks in addition to one realistic render per every material that can be applied to it.

Other applications offer to the user the possibility of uploading photographs to the system [29, 14]. In order to perform the customization of the environment, some actions are required by the application. In most of the analyzed virtual simulators, the user has to point over a middle tone in the area of the object to allow to cut off the area where the configurable object appears [6]. These applications usually require that the uploaded photography have some especial characteristics, such as a white or clear background where the object appears highlighted. Once the pixels that cover the configurable object have been identified, their color is changed by the finish chosen by the user.

More advanced applications, such as fractal graphics [5], build a 2D mesh over the target object. The user draws the contour of that object creating control points that finally are jointed in a Bezier curve [12,18]. A 2D mesh is built in real time over the demarcated area. Finally, texture mapping techniques are used to map the material selected by the user over this mesh. This solution requires more user interaction than the ones presented before. The main disadvantage is that the good quality of the results depends on the ability of the user defining this zone.

Based on images generated by rendering 3D environments Some virtual simulators obtain their environments from renders generated from 3D modeled scenes. Photorealism is obtained adjusting the lighting and material characteristics. This way of obtaining the set of images are easier and cheaper than to photograph real scenarios: only a computer and some modeling software are required [4]. Moreover, most of the render engines, such as V-Ray, Mental Ray, Arnold, Renderman, ... offer the possibility of rendering as separate images different elements of the lighting: direct, indirect or global illumination, or even some material characteristics: diffuse, specular, reflection, refraction, etc. These render elements allow to perform image composition [13]. This is a technique that has been widely used by the artists due to they can later control the final render: changing some parameters in the used image editing software, such as, for example, Photoshop or Nuke, allows to obtain different visual results. This is a very popular solution, because working with render elements separately makes it possible to have a greater control, freedom and ease to post product the final image [11].

Nevertheless, the render element that stores the diffuse characteristic of the material has to be managed with the same processes as the described in the previous section. Also alpha masks have to be applied to this image to separate the object to customize from the rest of the scene. This diffuse render element will be built from some snippets obtained from other renders, where the selected finish appears. Finally, the image editor software adds to the obtained diffuse element the rest of render elements adapting some visual parameters for each one and, as a result, the final image is achieved. The combination is performed by applying operations pixel by pixel, that vary according to the involved render element and the desired result. This technique performs very realistic images, because the render elements that store lighting and material characteristics have been carried out managing the render engine in advance.

However, its main disadvantage is the high storage cost. Besides all these render elements, the method requires, as in the previously described ones, two alpha masks for every area in the image that can be customized in addition to one realistic render per every material that can be applied to it. Moreover, an administrator user can not add new variations of the rendered environment in an easy way. If one new finish is required and has to be added to the application, the realistic image has to be rendered, together with the two alpha channel masks, if it is a new area. This fact makes that adding a new finish cannot be done by a user without any knowledge of 3D modeling programs.

2.2 Virtual tile designer

Tile factory is characterized by the large amount of models that offers and the assortment of sizes per tile. Besides this, there are also many dispositions a tile that can be shaped on the walls or floors. So, it is usual that customers want to see the final result of their choice in a scenario similar to the one they want to remodel.

The most of the virtual applications that deals with ceramic add the requirement of choosing the disposition of the selected tiles: users have to select the scenario, the area to customize, the tile and, finally, the tiling disposition [25, 26, 7]. Usually, some patterns are available in the virtual designers, and they condition the way tiles are distributed on the selected areas. Finally, the gasket that appears between the tiles are other requirement in some applications. This grout that frame the tiles can be configured, so users can select their thickness or color.

The analyzed applications use different methods to achieve the change of the tiles or disposition. Some of them add a new realistic image to the data stored per every possible collocation of a tile model on the wall or floor. This fact makes that the application considerably increases the storage cost: a same tile can be manufactured in different sizes so an image has to be included per each size to take this possibility into account.

Other applications are based on texture mapping techniques [32,8]. They include some masks that make it possible to map some tile dispositions on the surfaces. These masks have been generated in the 3D modeling software. They store the mapping coordinates (u, v) in the red and green channel so the texture can be applied to the final image extracting these coordinates. The image of the tile is the texture to map on the customizable wall or floor that appears in the image.

One texture mapping mask is required per every possible disposition, so the storage cost is reduced in regard to the method analyzed before. Only one beauty or final render is required per every tile in addition to the mask images that reflects the possible dispositions of it. However, if a new size is introduced in the application for a tile, a new disposition mask has to be generated and added to it. This fact makes that the updating of the tile models requires of the intervention of a 3D expert that reproduce the new tile disposition.

The application presented in this paper adapts the analyzed techniques and improves them resolving their main problems. It is based on images obtained from rendering 3D modeled scenes. Moreover, some render elements are generated from every modeled environment in order to adapt and change the characteristics of the lighting and materials on demand. The disposition of the tiles is calculated by the application: the tiling is performed taking the real measurements of the objects involved into account. This technique considerably reduces the amount of images managed by the application and makes it possible to upload tiles of any size to the virtual simulator, without the necessity of rendering a new image.

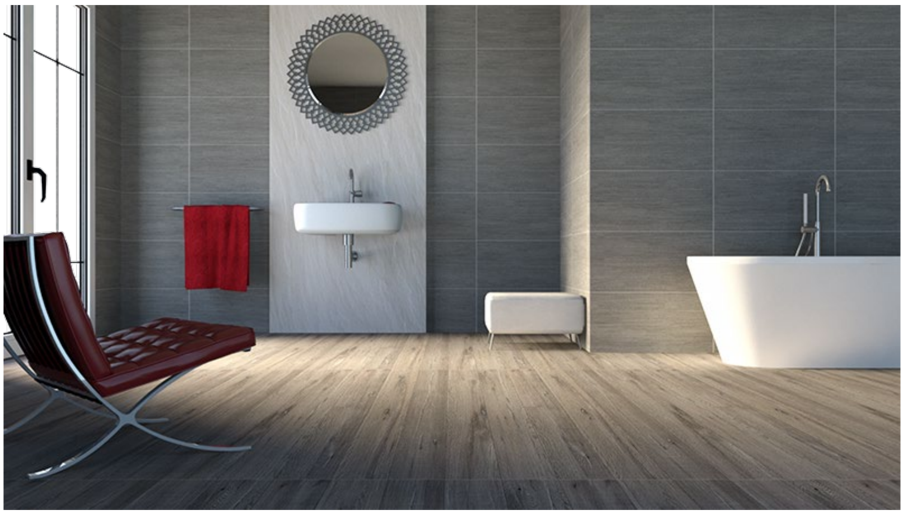


Fig. 1: An example of a 3D scene used to test the presented method.

3 Image elements

The developed application has been addressed to ceramic, so the 3D modeled scenarios represent bathrooms, kitchens or other rooms where tiles usually appear on the walls or floors. In this work, the different scenes have been modeled representing all the objects and elements with real-world measurements. This fact allows us to obtain a realistic render when physically lighting is applied. This kind of lighting produces that the final render simulates a photorealistic ren- der, as Fig. 1 shows.

Rendering different illumination components separately makes it possible to simulate a post-production process [21, 19]. Then, once the 3D scene has been modeled and accurately illuminated, different elements based on the lighting and material components are rendered. After analyzing them, the final image is decomposed in the following rendering elements:

$$FinalImage = Lighting + Reflection + Diffuse \tag{1}$$

The lighting render includes information from direct and indirect lights in the scene. Also shadows are included in this render (Fig 2).

Other render element that is separately generated is the reflection one. Different kinds of reflections have been considered, depending on the materials applied to the 3D objects. In the presented virtual simulator, users have the possibility of changing the material of some parts of the image on demand, so the reflections in these changing areas must vary according to the new assigned material. Different levels of reflections have been rendered and stored in images in order to apply the appropriate one to the material chosen in real time by the user.

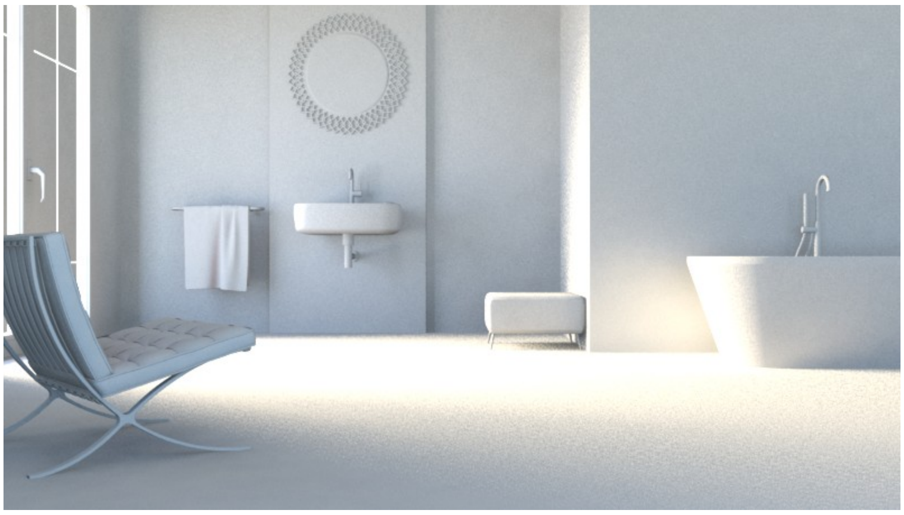
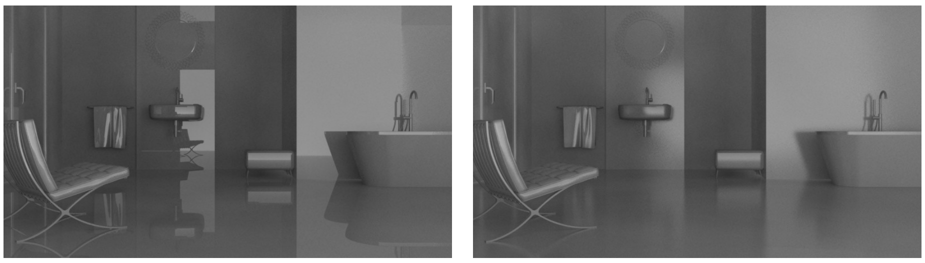


Fig. 2: Lighting render component.



(a) Glossy reflections.

(b) Diffuse reflections.

Fig. 3: Reflection render elements.

These images are obtained varying the glossy component of the applied material in the 3D scene. Fig. 3 shows two examples of reflections. The glossy reflections can be appreciated on the Fig. 3(a) and diffuse reflections of the same scene can be observed in Fig. 3(b).

The last render element that is required to compose the beauty or final image is the diffuse one. The diffuse component represents the basic color of the objects. This is addressed to render this part of the materials, where only plain colors appear. Some processes have been defined to obtained this element that will be the base over the rest of render elements will be applied. The processes have been divided in three steps: performing the tiling disposition of the walls and floor, calculating (if are required) the gaskets between tiles and, finally, adding the diffuse component of the atrezzo to compose the diffuse element.

3.1 Tiling disposition

The possibility of adding tile models of any size and quickly checking how they look when are tiled on the walls or floors, has been one of the main requirements of the presented application. In a real scenario, the number of tiles that maps a surface depends on the dimensions of that surface and the tile to be mapped. In the presented method, the same restrictions have been considered. In order to simulate this process, the application takes into account the real measurement of the modeled walls or floor and the real size of the tiles to map on them. On one hand, the measurements of the surfaces to be mapped are obtained by the application from the rendered images. On the other hand, the dimensions of the tiles are data introduced to the system (*tileSize.x*, *tileSize.y*).

After establishing a point on the surface that is considered as a starting point for tiling (0, 0), eg the lower left corner of the wall, the application requires knowing the distance of the pixel that is been analyzed with respect to that starting point. This distance (*position.x*, *position.y*) is calculated and stored in centimeters, because also the dimensions of the tile have been stored in these units. Once they are obtained, it is easy to calculate the point of the tile (*tilePos*) that has to be mapped on this pixel. It is obtained by performing a simple operation

$$tilePos = mod(position, tileSize) \qquad (2)$$

However, the application has to transform these centimeters in *uv* coordinates, *uvTile,* that have the values in the range [0,1]. This is achieved performing the operation:

$$uvTile = tilePos / tileSize; \qquad (3)$$

Every tile has to be analyzed and processed according to its size to obtain the appropriate mapping coordinates. Then, the procedure that simulates the real tiling on walls or floors described before has been designed and implemented taking advantage of the GPU programming. The presented implementation analyzes all the pixels of the customizable surface and establishes the pixel color according to the coordinates of the tile texture that has to be mapped. This process is performed in parallel to all the pixels in the area to customize.

Following the described process, the application has to calculate the distance of the pixel to the one considered as origin of tiling. To obtain this datum, some maps have been generated to make easier to measure the customizable surfaces.

Initially, an image that makes it possible to obtain the distance in centimeters to the origin of the tiling has to be mapped on the customizable surfaces. This image has to store the (*position.x, position.y*) values. Considering that the initial point of the measurement is the low-leftmost pixel, a RGB image has been designed where the red channel stores the *position.x* value and the green channel, the *position.y*. The image has been created of 256x256 resolution assuming that its measurement is 50x50 centimeter in the scene (Fig. 4).

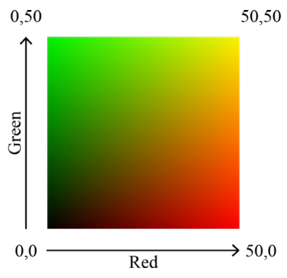


Fig. 4: Base texture of 256x256 resolution, equivalent to a square of 50x50 cm.

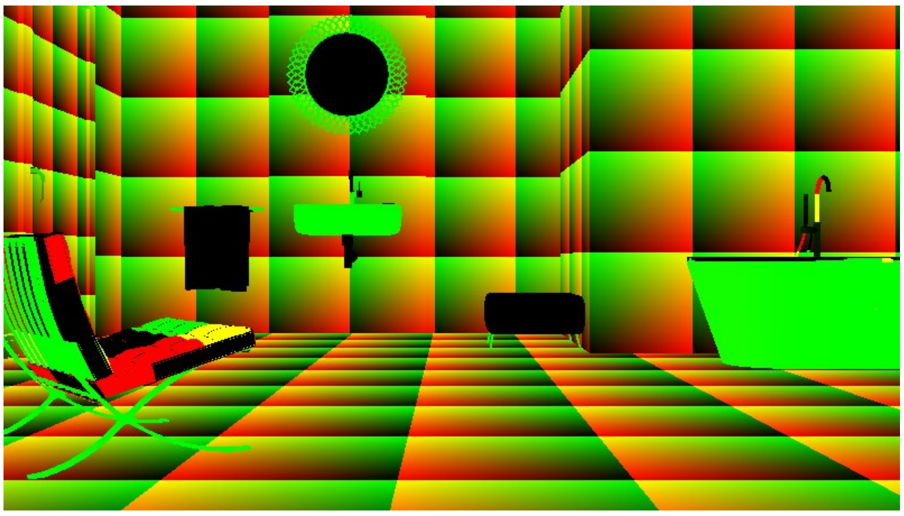


Fig. 5: Image obtained after mapping and tiling this image on the scenario.

The red and green channels store the distance of the pixel to the origin, from 0 to 50. The precision of this distance is 50/256 cm (*measurement*/*resolution*), i.e., $0, 19$ centimeters.

This texture is mapped and repeated on the customizable surfaces of the 3D scenario using real-world coordinates, so the RGB information is available in the rendered image. As the tiling has been performed using real-world coordinates, every pixel on the image has available two measurements in the range $[0, 50]$ centimeters: the one stored in the red channel provides information about the width of the surface, and the one in the green one provides information about the height. This render is shown in Fig. 5. In order to obtain realistic images, objects that are considered atrezzo in the scene reflect the texture mapped on the walls and floor. These data will be considered to compute the highlights in the case of the objects are reflective.

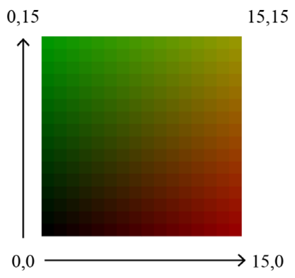


Fig. 6: Detail of the design of the texture that measures the meters in the image. It can cover up to 8 meters.

The measurements that are available in the image provides information of the centimeters, but to complete the process, other texture has been generated to compute information about meters. The second map has been created to manage this unit of measurement and to obtain how many meters there are from the analyzed pixel to the origin of the customizable surface. The purpose of this image is to know how many images of 50 cm have been mapped on the surface, both width and height. Then, the new map has been built representing areas of 256x256 (same resolution than image shown in Fig 4). with the same number in the red and green channels. It varies storing from 0 in the first area to 15 in the last one in the red and green channels (Fig. 6). Considering that every 256x256 area covers $0,5$ meters, the new image has been designed to measure up to 8 meters. That is enough for the kind of environments are been represented in the presented virtual simulator. Then, the resolution of this new map is 4096x4096.

Other image of the scenario is rendered by mapping this last image on the walls and on the floor of the 3D scene, where the tiles can be applied (Fig. 7). This render has been obtained maintaining the real-world coordinates in the mapping process and the same starting point for tiling. Also the objects that appear in the scene have to reflect the textures mapped on the wall and floor.

Reading the value red and green of every pixel, the distance of it to the origin of the wall or floor can be computed in meters, and the measurement represents real-world measurements. In order to compute the position of a pixel, our application reads the red and green value of the image shown in Fig. 7, and transform this value to centimeters.

Let *tcBig* be the data where this image is stored. As each considered area covers 50 cm, i.e., 0,5 meters, the performed operation is as follows:

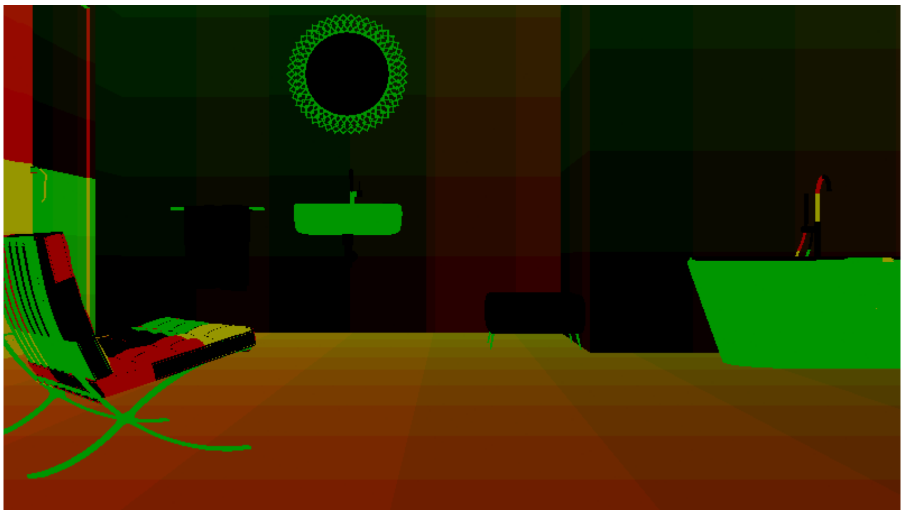


Fig. 7: Render that allows to obtain the measurements of the surface to be per- sonalized in meters.

$$tcBigm.xy = (tcBig.xy) * 0.5; \tag{4}$$

Finally, in order to give more precision, the same channels of the image shown in Fig. 5 are read for the same pixel. As the data that are retrieved represent centimeters, the previous measurement has to be converted to this unit (Equation 5).

$$tcBigm2cm.xy = tcBigm.xy * 100.0; \tag{5}$$

If the image that stores the centimeter information (Fig. 5) has been previously stored in *tc*, the operation that obtains the final measurement is shown in Equation 6.

$$(position.x, position.y) = tcBigm2cm.xy + tc.xy; \tag{6}$$

Once the application has obtained the position of the pixel (*position.x, position.y*), the tiles have to be processed in order to obtain the mapping coordinates. Every tile has determined measurements, depending on the model of the tile and, even for the same image, different dimensions can be managed by the application. An example is shown in Fig. 8.

Once the distance of the pixel to the starting tiling point is known, the exact coordinate of the tile that has to be textured on it is easily calculated. Finally, these measurements are converted to mapping coordinates and the problem is solved in real time by simple mathematical operations, as it was said before (Equation 2, 3).

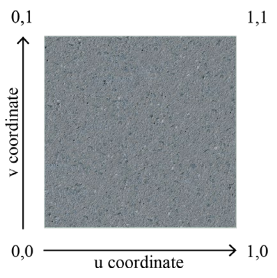


Fig. 8: A tile used in our application.

The pseudo-code that develops this process is shown in Algorithm 1. The function *CalculateTileCoordinates* is evaluated per every pixel in the image, *pixelPos*, taking the values red and green of it. Also, the function requires the dimensions of the tile, *tileSize*, that has been selected to map: the height and width measurements.

Let *ImageCM* be the mask that measures the centimeters and *ImageM* be the image that measures the meters. Both of them are evaluated in the position of the pixel (*pixelPos)* (lines 1-2). Once the data retrieved are converted to centimeters (lines 3-4), the distance of this point to the starting mapping point of the wall or floor is obtained. Finally, knowing the measurement of the tile to map, it is easy to calculate the mapping coordinates to be taken into account, *uvTile,* (lines 5-6).

Algorithm 1 Calculating the *uv* mapping coordinates.

```
//every 256x256 area in the ImageM render corresponds to 50 cm
DEF $m2cm \leftarrow 50.0$
procedure CalculateTileCoordinates (pixelPos, tileSize)
    //upload the images that allow to measure the walls and floor
1   $vec4\ tc \leftarrow texture2D\ (ImageCM,\ pixelPos)$
2   $vec4\ tcBig \leftarrow texture2D\ (ImageM,\ pixelPos)$
    //calculate the distance to the origin in cm where the pixel is situated
3   $vec2\ tcBig2cm \leftarrow tcBig.xy \times m2cm$
4   $vec2\ position \leftarrow (tc.xy + tcBig2cm)$
    //calculate the point of the tile that has to be mapped on the pixel (in cm)
5   $vec2\ tilePos \leftarrow position \bmod tileSize$
    //changing the position from cm units to uv mapping coordinates
6   $vec2\ uvTile \leftarrow tilePos/tileSize$
    return uvTile
end procedure
```

This algorithm has a constant computational cost [20]. The input parameters correspond with fields that stores two values: on one hand, the horizontal and the vertical position of the pixel to analyze and, on the other hand, the dimensions of the tile to map on the chosen surface. The location of the pixel is used to obtain the RGB values in the textures *ImageCM* and *ImageM*, that helps to calculate the real distance of the pixel to the mapping starting point. Then, mathematical operations are performed to obtain the final result. All of them have a constant cost, so the final theoretical computational cost is also constant.

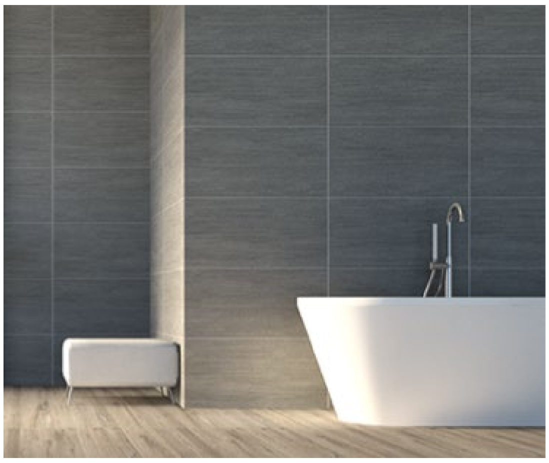


Fig. 9: Detail of the gaskets on the wall.

a. Gaskets of the tiles

In the tile industry, the gaskets between the different tiles are really important. Some tile models have in their final disposition this complement, so these gaskets have to be evident in the render. They help to visually show the size and collocation of the tiles in the case they are required. Gaskets are also computed in real time in the presented method.

Taking the mapping coordinates into account, it is determined if the evaluated point is a tile border and it is part of the gasket. Then, it has to be painted lighter than the rest of the tile. The gasket color can be chosen by the user of the application. However, that color is mixed with the color of the tiles, showing a smooth transition, as can be seen in Fig. 9. In this case, the color of the gasket has been initialized to (0.1,0.1,0.1) because when these values mix with the tile color, a subtle lightening of the final color is produced.

Algorithm 2 Calculating the gaskets between tiles.

```
procedure GasketCalculation(vec2 uvTile, vec2 tilesize)

    //obtaining the mapping coordinates of the left-most and top-most vertex of
    the tile

1   vec2 tcTileLeft ← CalculateTileCoordinates (vertexOntheLeft, tilesize)
2   vec2 tcTileTop ← CalculateTileCoordinates (vertexOntheTop, tilesize)

    //checking if the current pixel is part of the gasket (very close to the left or
    to the top)

3   if ((abs (uvTile.x - tcTileLeft.x) > 0.2)) || (abs (uvTile.y - tcTileTop.y) > 0.2))
4     return vec3(0.1, 0.1, 0.1)
5   else
6     return vec3(0.0, 0.0, 0.0)
7   endif

end procedure
```

The implementation of this process is shown in Algorithm 2. The implemented function, *GasketCalculation*, returns the values that have to be added to the pixel color that is evaluated. In order to check if the evaluated pixel is part of the gasket, only the sides left and top of the tile are going to be considered, to avoid painting a gasket twice, because of the regular disposition of the tiling. First of all, the mapping coordinates of the left-most vertex, *vertexOntheLeft*, and the top-most vertex, *vertexOntheTop*, are obtained (line 1-2). Then, if the mapping coordinates of the pixel, *uvTile*, is closer than a certain distance to the values previously obtained (line 3), this point is considered part of the gasket.

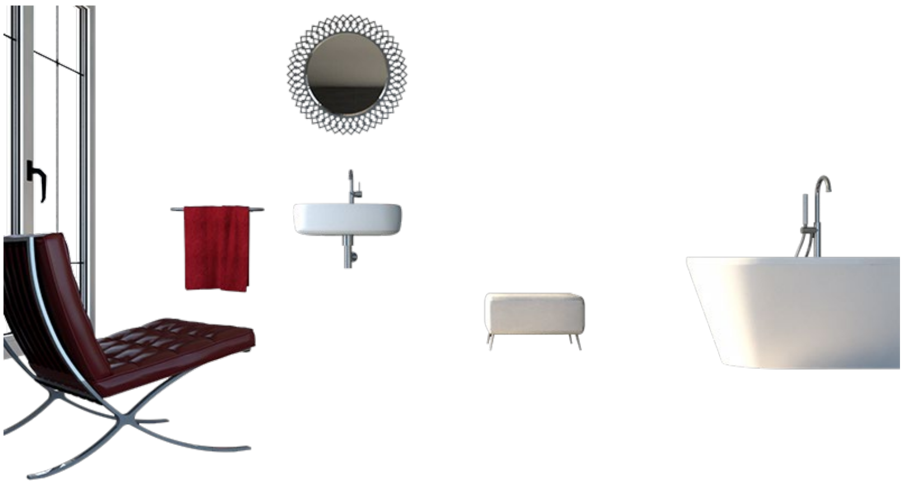


Fig. 10: Diffuse render element of the atrezzo.

The implemented function returns the color that has to be added to the pixel color to simulate the line surrounding the tiles (line 4). The distance that has been considered to be the thickness of the gasket has been 0.2. However, if thicker gasket is required, this number can be increased.

The computational cost of this algorithm is constant. The function *CalculateTileCoordinates,* called in this code, has a constant cost, as has been previously analyzed. Apart from this, the algorithm includes only two comparisons among two values. All of this determines the constant cost of the code showed in Algorithm 2.

3.2 Composing the atrezzo

The diffuse element is composed by combining different layers independently processed: the configurable areas and the attrezo. On one hand, the images of customizable surfaces are generated on demand, taking into account the tile model selected by the user and following the process explained before. The presented application considers two customizable surfaces: a wall and the floor. However, the amount of surfaces that can be personalized is easily scalable.

On other hand, the diffuse components of the materials that form the atrezzo of the scenario (Fig. 10) are stored in the database. This image has to store some transparent areas, so the file format that has been chosen to store all the required masks is png format.

6 Image composition

The diffuse layer mainly conditions the final appearance. Once this one has been obtained, the developed method adds the lighting information and the reflection characteristics of the materials. In order to accelerate the processes, the advantages in GPU has been taken into account [15]. Then, the implemented method has been developed with shader programming, performing all the operations at pixel level.

Any 3D modeling software can be used that has the required rendering utilities and allows us to obtain the different images that will be finally combined to obtain the target composition. The implementation has been developed following the GLSL specification, performing all the operations per-pixel to compose the final image. It takes into account multiple base textures and

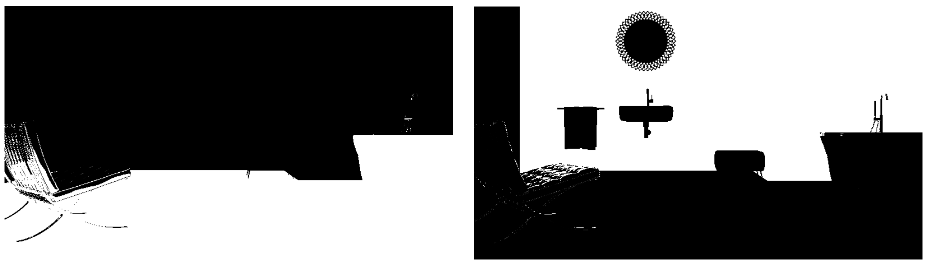

(a) Unmasking the floor. (b) Unmasking the wall.

Fig. 11: Masks used to discriminate some areas in the image.

combines them with the other components: masks, reflections and lighting textures to produce the final color.

The user of the virtual simulator determines the area to customize only clicking over the image in the area where this surface appears. In order to determine the surface that has been selected, some masks have been rendered from the 3D scene and have been included in the application. These masks will be used in the merging process and they have to be combined with the final tiling disposition image to extract the area in the image covered by the customizable surface. The masks have been rendered as an alpha mask: the parts of the image to be discarded are rendered in black color and the areas that have to remain, rendered in white. They are managed as simple alpha channels in the post-production process.

Fig. 11 shows the two different masking images used in the virtual scene of the example. The walls and some parts of the atrezzo will be discarded if mask shown on Fig 11(a) is used. The image shown on Fig. 11(b) masks the floor and the atrezzo. The windows and holes that appear on the walls have been rendered in black in both images to avoid of changes.

The pseudocode of the fragment shader that makes it possible to perform the final composition is shown in Algorithm 3. Let *p* be a screen position in the image where the final color in the image is going to be created. Regarding the tiles, let *floorTileWidth* and *floorTileHeight* be the width and height of the tile to be mapped on the floor and *wallTileWidth* and *wallTileHeight* be the size of the one to be tilled on the wall. The images of both tiles are stored in *floorTileImage* and *wallTileImage*. Let *groundMaskImage* and *wallMaskImage*be the images where the discriminatory masks are stored. Moreover, the diffuse component of the atrezzo that is in the scenario has been rendered in the image *atrezzoImage*. Finally, let *gasketsFloor* and *gasketsWall* be a boolean that determines if the tile model requires that gaskets are visible on the surface of the floor or the wall.

As the processes in every position in the image is performed in parallel, the developed algorithm calculate the tiling for the wall and the floor simultaneously. Also, if the gasket is required for the model of tile selected by the user, the final image where they are drawn are computed. Then, the black and white masks are applied to these images in order to extract only the areas that are necessary. Both resulting images are combined with the one that contains only the atrezzo in order to compose the diffuse element. Finally, the stored render elements that contains the global illumination and the reflection one are combined with the diffuse component to obtain the final color.

Algorithm 3 Calculations performed per pixel at GPU level.

```
FOR every $p$

//Calculate the tile mapping coordinates $(u,v)$ of the pixel $p$ according to the dimensions of
the tile used for the floor and the one used for the wall

$vec2\ tcTileFloor \leftarrow CalculateTileCoordinates(p,\ vec2(floorTileWidth, floorTileHeight))$
$vec2\ tcTileWall \leftarrow CalculateTileCoordinates(p,\ vec2(wallTileWidth, wallTileHeight))$

//Obtain the information of the RGB color for the pixel

$texfloor \leftarrow texture2D\ (floorTileImage,\ tcTileFloor)$
$texwall \leftarrow texture2D\ (wallTileImage,\ tcTileWall)$

//Read the pixel information in the alpha mask images

$maskFloor \leftarrow texture2D\ (groundMaskImage,\ p)$
$maskWall \leftarrow texture2D\ (wallMaskImage,\ p)$

//Calculate the gaskets for the floor if they are required

if $gasketsFloor$ then
  $texfloor.xy \leftarrow texfloor.xy +$
        $GasketCalculation\ (tcTileFloor,\ (floorTileWidth, floorTileHeight))$
endif

//Calculate the gaskets for the wall if they are required

if $gasketsWall$ then
  $texwall.xy \leftarrow texwall.xy +$
        $GasketCalculation\ (tcTileWall,\ (wallTileWidth,\ wallTileHeight))$
endif

//Compose the diffuse render element

$floorColor \leftarrow texfloor \times maskFloor$
$wallColor \leftarrow texwall \times maskWall$

//Read the information of the diffuse atrezzo color for the pixel $p$

$atrezzoColor \leftarrow texture2D\ (atrezzoImage,\ p)$

//Compose the diffuse component for the pixel $p$

$diffuseElement \leftarrow floorColor + wallColor + atrezzoColor$

//Obtain the information of the rest of render elements in the pixel $p$

$global\_Ilumination \leftarrow texture2D\ (\ global\_IluminationImage,\ p)$
$reflectionElement \leftarrow texture2D\ (\ reflectionIluminationImage,\ p)$

//Calculate the final appearance of the pixel

$finalColor \leftarrow diffuseElement \times global\_Ilumination \times reflectionElement$

//Returns the result of the fragment shader

$glFragColor \leftarrow finalColor$

END FOR
```

Algorithm 3 codes the main function of the presented method. The user of the application clicks on the area of the image where the surface to configure appears, the image of the tile and finally, the size of that tile. This code is performed per every pixel of the image, creating a thread per every one of them in GPU. Let *n* be the number of pixels of the images. This value can vary according to the resolution of the images that are used in the application (*with* x *height*). Finally, analyzing the theoretical computational cost [2] [20], it can be said that the computational cost of the presented application is *O(n),* i.e., it depends on the resolution of the images (or number of pixels) that have been previously rendered and used for performing the computations.

5 Evaluation and Results

In order to evaluate our virtual simulator, three scenes have been modeled using 3DS Max 2017. In both of them, a wall and the floor can be customized, and 30 tiles with 3 different sizes have been included in the application that can be chosen by the user to personalize the floor. The same amount of tiles with also 3 sizes are available for the walls.

Every image rendered from the 3D scene has been stored in png format with a resolution HD, 1920*x*1080 with 72 ppp. The size of each rendered image is 606 KB. The tile images have been stored in jpg format, with a resolution of 512*x*512 with 72 ppp. The size of every one is 130 KB.

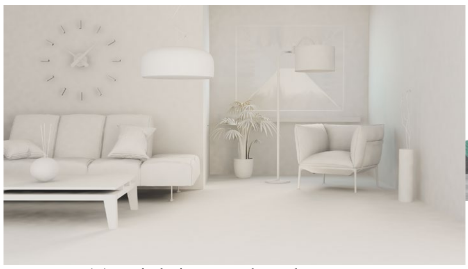
The other modeled scenes represent a living room and a terrace. The images generated to perform the test for the other 2 scenes are shown in Fig. 12 for the living room and in Fig. 13 for the terrace.

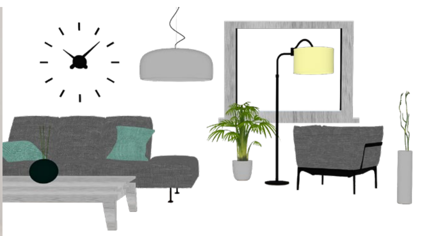
Due to the fact that the application has been addressed to be available on Internet or app stores, one of its main advantage is the low storage cost compared with the most popular solutions. The presented method has been compared with the most popular one based on images, as is the used in [22]. We have called this technique the Rendered Images Method, because it renders all the possible finishes in advance, applying masks to obtain the desired combinations.

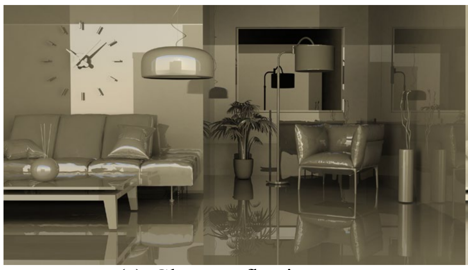
The data required to perform the virtual configurator where the three modeled environments appear are analyzed in Table 1. As it is said, the final user can choose among a set of 30 tiles of 3 different sizes for the walls, i.e., 90 different models, and other different 90 models for tiling the floor.

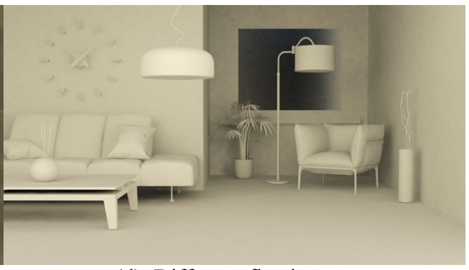
In the case of adding a new tile model that is manufactured in two sizes, two images of 1080*x*1920 resolution beauty renders have to be uploaded by the Rendered Images Method. However, only the image of the tile has to be uploaded in the case of the presented method, because the two sizes used the same image. Tile factories usually sell hundreds of tile models, so this fact makes that the more models are loaded the greater the difference in the storage cost of the methods.

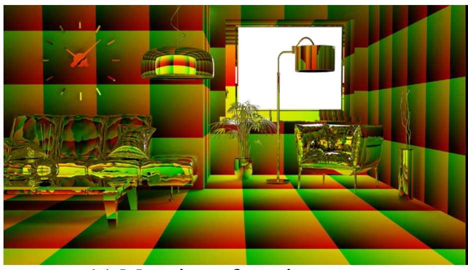
Other main advantage of the presented method is the saving in the render time of the images. Depending on the configuration of the global illumination and on the characteristic of the assigned materials, the time for rendering an image can take some hours. The time of mapping the new texture has to be added to the time spent in obtaining a new image. In the presented method, the

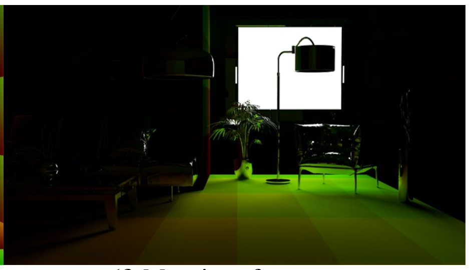

(a) Lighting render element.

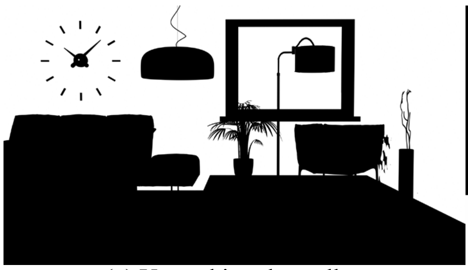

(b) Diffuse render of atrezzo.


(c) Glossy reflections.


(d) Diffuse reflections.


(e) Mapping of centimeters.


(f) Mapping of meters.


(g) Unmasking the wall.

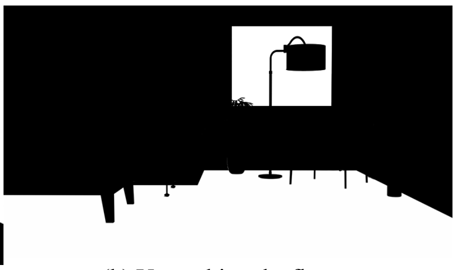

(h) Unmasking the floor.

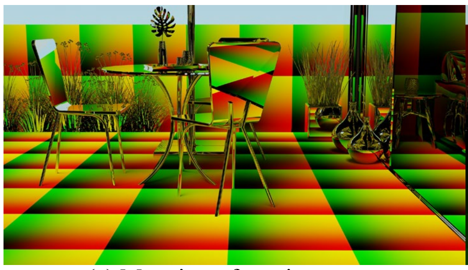
Fig. 12: Renders to configure the modeled living room.

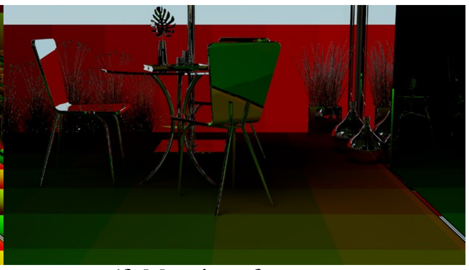
(a) Lighting render element.

(b) Diffuse render of atrezzo.

(c) Glossy reflections.

(d) Diffuse reflections.


(e) Mapping of centimeters.


(f) Mapping of meters.

(g) Unmasking the wall.

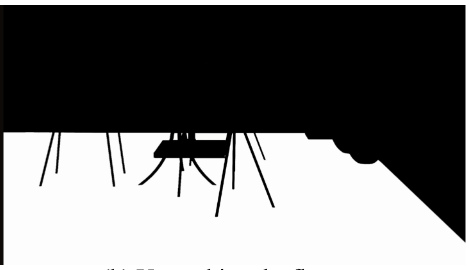

(h) Unmasking the floor.

Fig. 13: Renders to configure the modeled terrace.

Table 1: Storage comparison between both methods.

Rendered Images Method				
Type of images	Number of images	Data size/scene	Final size	Data stored
Beauty images	90 images/scene	53, 27 MB	159, 78 MB	163, 33 MB
Mask images	2 images/scene	1, 18 MB	3, 55 MB	

Presented Method				
Type of images	Number of images	Data size/scene	Final size	Data stored
Beauty images	0 images	0 MB	0 MB	21, 8 MB
Mask images	2 (alpha mask)/scene 2 (mapping mask)/scene	2,36 MB	7,10 MB	
Render elements	4 images/scene (lighting+diffuse+2*reflections)	2,36 MB	7, 10MB	
Tile images	60 images	7,6 MB	7, 6MB	

render elements that are required are rendered in parallel: the renderengine extract every element in the same rendering process.

Finally, regarding the realism obtained with the presented method, some new finishes are shown from Fig. 14 to 16 where different tiles have been mapped. The one mapped on the wall on the living room does not require to add gaskets to the surface.

7 Conclusions and future work

The design of a virtual simulator addressed to tile factories has been presented in this paper. Different scenarios can be quickly customized by changing the tiles that map the floor or the wall. The user of the application selects a customizable surface, a tile (image and size) and in an almost negligible time the surface changes its appearance. The virtual simulator has been implemented taking advantage of the GPU, so shader programming language has made that the computation time is almost insignificant.

The presented method simulates the real world tiling. Depending on the real-world measurements of the configurable surface and the size of the tile model to map on them, a pixel is represented with a determined color. Analyzing some mapping masks previously generated, the method obtains the distance ofthe pixel to the mapping starting point. This measurement, converted to centimeters, makes possible to determine the mapping coordinates of the tile image, so the color of the pixel is obtained.

Moreover, some render elements that stores different image characteristics are also stored in the application. The combination of these elements with the diffuse one obtained by the shaders, makes it possible to obtain very realistic images. Varying the combination of the render element allows the user to get different finishes of the same scenario, making the application very versatile.

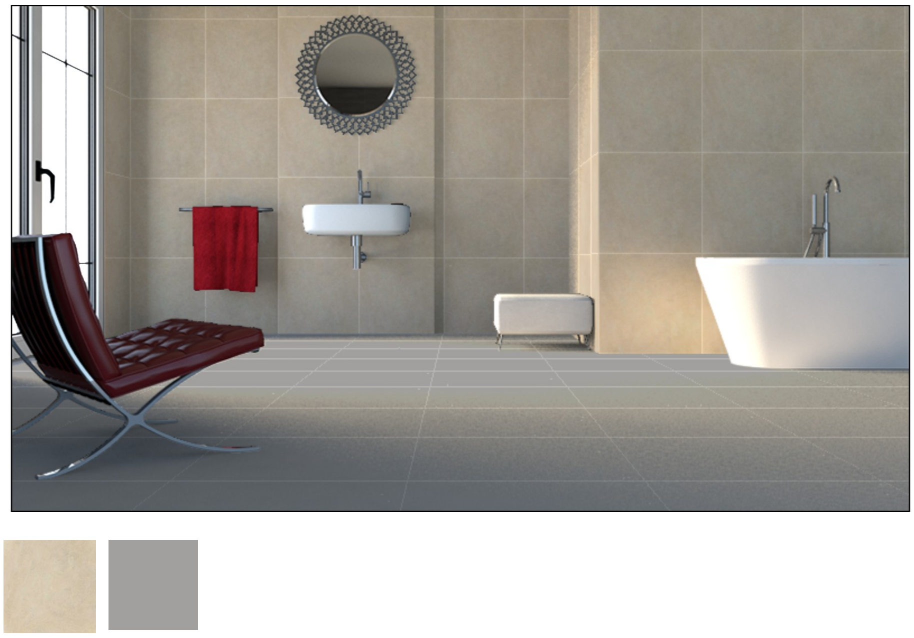


Fig. 14: Final composition of the bathroom with the tiles that are shown.

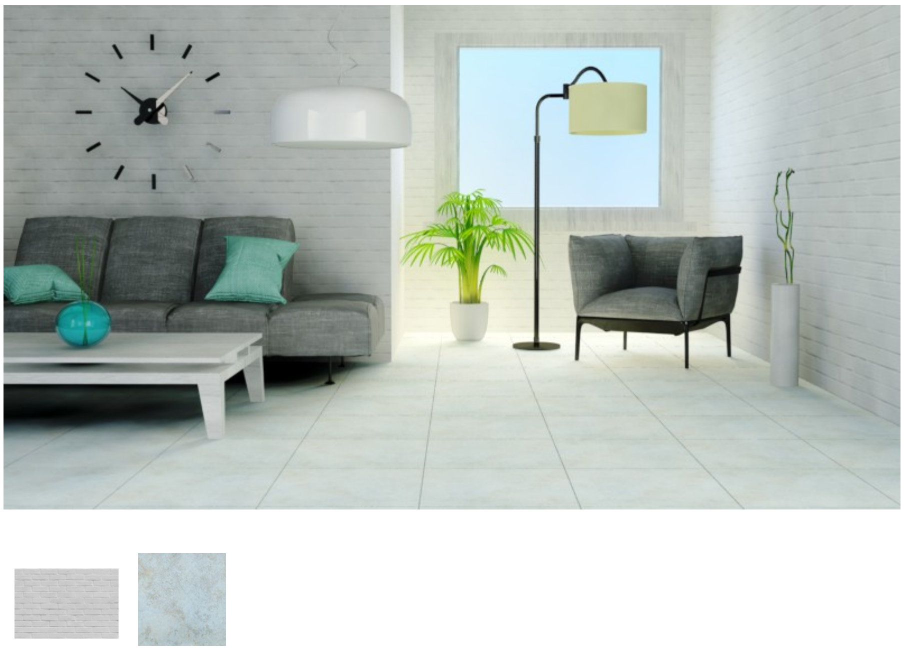


Fig. 15: Final composition of the living room, using the tiles shown in the figure.

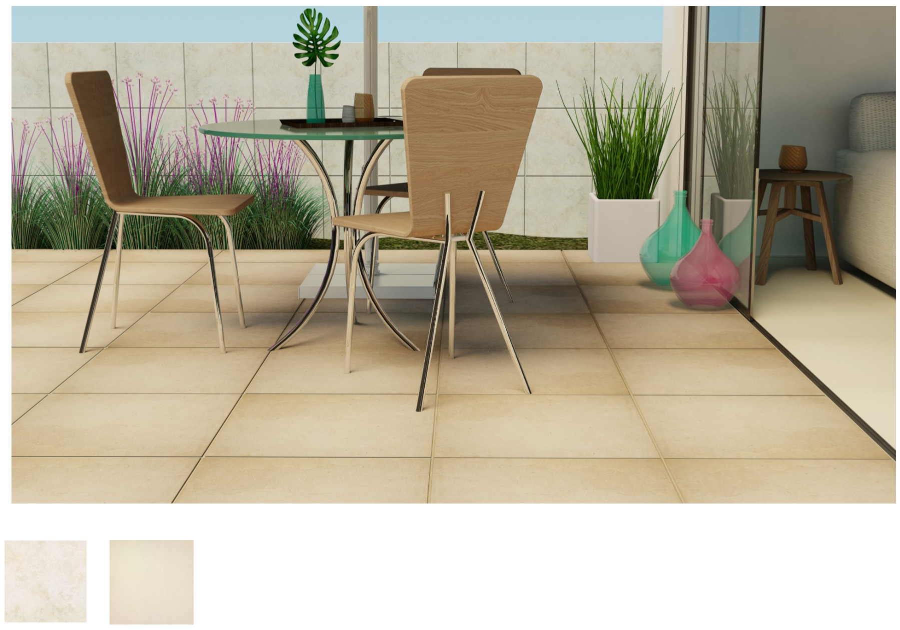


Fig. 16: Final composition of the terrace scene with the tiles shown.

Other advantage of the presented application is the ease of adding a new size for a tile model or even a new tile image. Only has to be indicated to the virtual simulator the dimensions of the new tile and the application will calculate the mapping coordinates for this new size and will show it mapped on the customizable surfaces when it is required.

Different fields can be improved in this virtual simulator as a future work. One line of improvement that we are currently working on is to visualize the environments in some VR devices, in order to achieve the immersion of the user in the scene. Some 360 scenarios have to be rendered instead of some images. The interaction will be performed by the devices that present some VR glasses, as the Vive Vr Glasses [30], that make it possible for the user to interact with the environment. The selection of the different tile models and the area to configure can be easily performed by them.

Acknowledgements This work was supported by the Spanish Ministry of Science and Technology (Project TIN2016-75866-C3-1-R).

References

1. Behr Paint Colours, http://www.behr.com/consumer_ca/colors/paint, Accessed October 2017
2. Card S K.; Moran, W P, Newell A (1980). The keystroke-level model for user performance time with interactive systems. Communications of the ACM. 23(7): pp 396–410

3. Dahan E, Hauser J.R (2012) The virtual customer, Journal of Product Innovation Management, 19(5):332-353
4. Ebert D, Kenton F, Peachey D, Perlin K, Worley S (2002) Texturing and Modeling: A Procedural Approach 3rd, 600, Morgan Kaufmann Publishers Inc. San Francisco, CA, USA
5. Fractal Graphics, http://www.fractalgraphics.com Accessed, October 2017
6. Gevers T, Smeulders A.W.M (1999) Color-based object recognition. Pattern Recognition, 32:453-464
7. Grupo Halcon, http://ambients.halconceramicas.com, Accessed October 2017
8. Heckbert P (1986) Survey of texture mapping, IEEE Computer Graphics and Applications, 6(11):56-67
9. Ikea Home Planner, http://www.ikea.com, Accessed October 2017
10. Khurana P, Sharma A, Singh S.N, Singh P.K (2016) A survey on object recognition and segmentation techniques, Proceedings of 3rd International Conference on Computing for Sustainable Global Development (INDIACom), pp 3822-3826
11. Lengyel J, Snyder J (1997) Rendering with Coherent Layers, Proceedings of the 24th annual Conference on Computer Graphics and Interactive Techniques, pp 233-242
12. Linares J, Santonja J, Chover M (2003) Realistic Image Mapping System and its Multimedia and Internet Integration, Industrial & Project Presentations, pp 27-34
13. Molnar S, Eyles J, Poulton J (1992) PixelFlow: High-Speed Rendering Using Image Composition, Proceedings of the 19th annual Conference on Computer Graphics and Interactive Techniques, pp 231-240
14. Moore B., http://www2.benjaminmoore.com/en-ca/for-your-home/personal-colour-viewer, Accessed October 2017
15. Pharr M, Fernando R (2005) GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation, Addison-Wesley Professional
16. Remolar I, Chover M, Quirós R, Gumbau J, Ramos F, Castelló P, Rebollo C (2010) Virtual Trade Fair: A Multiuser 3D Virtual World for Business, Proceedings of 2010 International Conference on CyberWorlds, pp 208-214
17. Rodda Color Visualizer, http://www.roddapaint.com/professionals/color/color-visualizer, Accessed October 2017
18. Santonja J, Linares J, Chover M (2002) An image mapping system for simulating ceramic tiles on real photographs, International Conference on Computer Vision and Graphics, 1:121-128
19. Schied C, Dachsbache C (2015) Deferred attribute interpolation for memory-efficient deferred shading, Proceedings of the 7th Conference on High-Performance Graphics, pp 43-49
20. Schnitzer S, Gansel S, Durr F, Rothermel K (2014) Concepts for execution time prediction of 3D GPU rendering. Proceedings of the 9th IEEE International Symposium in Industrial Embedded Systems (SIES), pp. 160-169
21. Soler C, Hoel O, Rochet F (2010) A deferred shading pipeline for real-time indirect illumination, Proceeding SIGGRAPH 2010 Talks, Article 18
22. Spark Vision Digital Showroom, http://www.spark-vision.com/digitalshowroom, Accessed November 2017
23. Sukanya C, Roopa G, Vince PA (2016) Survey on Object Recognition Methods, International Journal of Computer Science Engineering and Technology, 6(1):48-52
24. Sweet Home 3D, http://www.sweethome3D.com/es/index.jsp, Accessed October 2017
25. Tile Giant, https://www.tilegiant.co.uk/idesign/, Accessed October 2017
26. Tile planner, http://www.tileplanner.com/es/, Accessed October 2017
27. TitanLux, https://www.titanlux.es/en/inspiracion, Accessed October 2017
28. Trentin A, Perin E, Forza C, Increasing the consumer-perceived benefits of a mass-customization experience through sales-configurator capabilities, Computers in Industry, 65(4):693-705 (2014)
29. Valspar Painting, http://www.valsparpaint.com/en/explore-colors/painter/, Accessed November 2017
30. Vive VR Glasses, https://www.vive.com/, Accessed October 2017
31. Wang Z, Ziou D, Armenakis C, Li D, Li Q (2005) A comparative analysis of image fusion methods, IEEE Trans. Geosci. Remote Sens, 43(6):1391-1402
32. Xin L (2015) On Computing Mapping of 3D Objects: A Survey, ACM Computing Surveys (CSUR) Surveys, 47(2)