iSurfer

un Ray Tracer de superficies algebraicas en tiempo real para iPhone

Autores: Daniel José Azar
Damián Modernell
Cristian Prieto

Tutor: Andreas Daniel Matt

Tabla de contenido

Abstract

Este es el informe del proyecto final iSurfer de la carrera de Grado de Ingeniería informática en el instituto tecnológico de buenos aires. El trabajo es el primer graficador de superficies algebraicas en dispositivos móviles utilizando un Raytracer.

Introducción

Basado en el éxito de Imaginary, una exhibición itinerante de matemática originada en Alemania en el año 2008. La aplicación por excelencia en esta exhibición es el Surfer. El Surfer es una aplicación para graficar en tiempo real superficies algebraicas. Una superficie algebraica esta dada por el conjunto donde la ecuación polinomial en las 3 variables es igual a 0.

Una superficie algebraica tiene 3 dimensiones x, y, z. En cuanto al grado de la ecuación este se mide como el grado de un polinomio normal donde todas las variables independientes son tenidas en cuenta. Donde por ejemplo si tenemos $x^2+y^2*z=0$ el grado es 3.

En la actualidad los dispositivos móviles como los celulares y las tabletas alcanzan en conjunto mas unidades que las computadoras tradicionales. Estos a su vez han crecido en sus capacidades de procesamiento de datos a un nivel considerable.. Es por eso que recientemente se a podido generar distinto tipo de aplicaciones basadas en RayTracing en los mismos como es el caso de iGPUTrace. Pero esta potencia no es suficiente para el calculo de superficies algebraicas utilizando los métodos tradicionales, esto debe hacerse en la GPU para poder aprovechar todas las optimizaciones matemáticas de la misma.

Desafío

Para visualizar estas superficies existen dos métodos. El primero consiste en paramétrizar las ecuaciones y hacer una teselasión de la superficie. Esto consiste en generar triángulos con la forma de la superficie y eso es lo que se muestra. Este método tiene la ventaja de ser muy veloz, el problema de este método consiste en la precisión y los detalles de la superficie los cuales se pierden con el mismo. El segundo método es utilizar un Ray Tracer donde desde la cámara se lanzan los rayos, solo que en lugar de chocar contra objetos, se evalúa si la ecuación tiene una raíz en cada punto por los que pasa el rayo. Dicho método es muy lento ya que exige mucha

procesamiento pero tiene la ventaja de contar con la representación mas fiel que nos pueda brindar el dispositivo de la superficie.

El Ray Tracing se basa en el Ray Caster, lo que hacen es lanzar rayos desde el observador (la cámara) hacia la escena. Estos Rayos están compuestos por el origen y la dirección, al ser en un mundo 3D la ecuación contiene las 3 variables *x*, *y* y *z* *r=mt+b.* Es decir t es una variable unidimensional mientras que m es una constante en las 3 variables que representa la dirección y b es un punto en el espacio que representa a la posición de la cámara. Para la intersección en este caso se utiliza una grilla 3D, se paramétrica el polinomio original de X,Y y Z a t es decir X(t), Y(t) y z(t). Para el calculo de intersección se calcula si existen raíces en un intervalo [tmin ; tmax].

Para optimizar este proceso se utilizan algoritmos que permitan conocer si existen y donde se ubican las raíces. De no ser así habría que evaluar el polinomio para todos los puntos de la grilla por los que pase el rayo. Dependiendo del grado del polinomio resultante se pueden utilizar algoritmos directos:

- Grado 1: se resuelve linealmente.
- Grado 2: se utiliza la formula resolvente.
- Grado 3: se utiliza Cardano.
- Grado 4+: se utiliza Descartes.

Por un tema de Performance en el iPhone y los dispositivos móviles, es muy difícil utilizar la CPU para estos cálculos Matemáticamente intensivos. Por estos motivos se decidió utilizar OpenGL y la GPU de los dispositivos la cual esta mas optimizada para realizar estos cálculos y simplifica la parte de Ray Tracing.

OpenGL consta de cuatro partes, la interacción con la CPU, el Vertex shader, el Fragment shader y el Geometry shader. En los dispositivos móviles existen versiones distintas que las de las placas de video de las computadoras de escritorio. Las versiones móviles existentes son OpenGL 1.0 y 2.0, la diferencia principal radica en que el primero es una interfaz simplificada donde OpenGL se encarga de los shader y las luces. Mientras que 2.0 es donde podemos utilizar nuestros shader pero perdemos toda simplificación que existía en el 1.0. Para este desarrollo se utilizo OpenGL ES 2.0 donde se crearon Vertex y Fragment shaders.

El objetivo y desafío del presente proyecto es lograr implementar y adaptar el tipo de aplicación explicado anteriormente en dispositivos con menor poder de computo como los dispositivos móviles.

Especificación de requerimientos

Funcionales

EL proyecto iSurfer surge a partir del actual y vigente proyecto Surfer, el cual se encuentra disponible para PC y Mac.

El mismo consiste en los siguientes requerimientos funcionales:

- Poder generar superficies algebraicas.
- Permitir al usuario rotar la superficie utilizando la tecnología “touch” presente en los teléfonos inteligentes
- Permitir crear galerías y poder almacenar superficies generadas en las mismas.
- Publicar superficies en redes sociales “Facebook”
- Cambiar el color de las superficies al igual que el Surfer

Para una lista mas detallada de los mismos ver archivo adjunto.

No funcionales

Los requerimientos no funcionales incluyen:

- utilizar los mismos lineamientos técnicos presentes en Surfer, adaptados a la tecnología iOS para obtener la representación mas fiel de la superficie.
- Generación de superficies en un tiempo acotado, tolerable para la experiencia de usuario
- Rotación fluida de las superficies mediante la tecnología “touch” que dé una idea de rotación tridimensional de la misma
- Interfaz de usuario amigable, siguiendo lineamientos de Apple.

Para una lista mas detallada de los mismos ver archivo adjunto.

Arquitectura de la aplicación

La aplicación esta desarrollada tanto en el lenguaje nativo de iOS (Objective –C) como en C , C++, HTML5 y GLSL.
El patrón de arquitectura utilizado es MVC (Model View Controller) el cual es el adoptado por todas las aplicaciones iOS.
Los frameworks utilizados son

- UIKit (framework de componentes nativas iOS como botones, cuadros de texto, labels, etc.)
- socialNetwork (framework nativo para publicar en redes sociales)
- OpenGL ES 2.0 (framework de interfaz con el GPU)
- CoreGraphics(framework de adición de efectos a componentes nativos)

Para el almacenamiento de las superficies y demás información se utiliza una base de datos sqLite

La aplicación se instala en un dispositivo a través del Store de aplicaciones de Apple. La misma corre en un único hilo de ejecución en el procesador, dado que el mismo solo debe encargarse de la interfaz gráfica y acceso a la base de datos. La generación y

procesamiento de las superficies (que es el core del procesamiento) se realiza dentro del GPU a través de la generación de shaders que se ejecutan en el mismo.

Plan de avances

El proceso del proyecto se puede dividir en 6 grandes etapas.

Relevamiento de requerimientos

La primera etapa consistió en determinar si los requerimientos del proyecto eran factibles de ser implementados en un dispositivo iOS (iPhone o iPod touch), así como también determinar posibles restricciones propias del hardware. Para ello hubo una etapa de investigación de la arquitectura de un dispositivo iOS y de funcionamiento del GPU principalmente.

Aprendizaje de las herramientas

La segunda etapa, se baso en el aprendizaje de las herramientas de desarrollo y los distintos lenguajes de desarrollo a utilizar. A su vez se armo una serie de charlas donde cada integrante le explicaba a sus pares los distintos conocimientos.

Diseño de interfaz visual

Durante la tercer etapa, se procedió al diseño de la interfaz visual de la aplicación. Se tuvieron en cuenta para el diseño de la interfaz la forma de lograr una experiencia de usuario óptima, decidiéndose por ejemplo una interfaz visual simple en modo apaisado y siguiendo los lineamientos de interfaz de usuario de Apple.

Desarrollo

La siguiente etapa incluye el comienzo del desarrollo de la aplicación, tanto visual como de la lógica de generación de superficies algebraicas. Se comenzó simultáneamente con la interfaz visual, el parser de ecuaciones y la creación y acceso a la base de datos.
Luego se volcó el esfuerzo en la adaptación de los algoritmos utilizados en Surfer para La generación del ray tracing de superficies al lenguaje Objective – C.

Estabilización

Esta etapa fue la mas extensa y se enfocó en la solución de “bugs” y el correcto funcionamiento de todas las partes de la aplicación.

Rediseño

Por ultimo una vez que la aplicación funcionaba correctamente se procedió a hacer modificaciones de la interfaz visual y de flow de la misma. Esto abarco toda la aplicación.

Entorno de Desarrollo

Programar en iPhone

Para poder programar en iOS es necesario utilizar una PC de Apple. Primero se debe instalar el XCode que en la actualidad se puede descargar gratuitamente del App Store de OSX. Luego se deben descargar los simuladores del iPhone. Existen varias condiciones que una aplicación debe soportar para poder ser subida al App Store de iOS como puede ser el caso de contener cada imagen en su tamaño y en dos veces su tamaño para poder soportar pantallas retina.

GPU

La elección de la GPU se debió a la poca capacidad de procesamiento de estos dispositivos. Al realizar un Ray Tracer la GPU realiza automáticamente optimizada por hardware la creación de los rayos. A su vez esta optimizada para las operaciones matemáticas de luces y la mayoría de los cálculos que nosotros requerimos. El iPhone 3GS es el primer dispositivo de Apple que nos permite utilizar OpenGL ES 2.0 por lo cual la aplicación solo funcionara para los dispositivos mas nuevos. Mientras que la CPU de estos dispositivos suelen ser dual core o quad core, la Gpu puede tener hasta 64 hilos simultáneos.

Para programar la GPU hay que utilizar el lenguaje GLSL. Es un lenguaje de programación muy simple, similar a C. Al no utilizarlo para lo que esta diseñado hay que ingeniarse como pasar los datos a través del pipeline. Desde la CPU se le pasan constantes y puntos de vértices. Esto llega al Vertex shader que genera los rayos para cada pixel. Entre el Vertex shader y el Fragment shader la comunicación se realiza solo a partir de variables que se modifican entre pixel y pixel. El Fragment shader solo tiene como salida el color del pixel.

Este código al ser ejecutado fuera de la CPU no existe un debugger ni se puede imprimir mensajes a consola. La única forma posible de debuggear el código es pintando del Color X si pasa algo o Y en caso contrario. Para poder hacer esto hay que analizar bien el código y saber el input, lo que uno busca y cual es el resultado esperado en cada línea de código.

Documentación de Código

Para la documentación del código se decidió utilizar YuiDoc debido a la gran flexibilidad que nos permite al poder documentar dentro de las interfaces y el código. Esta documentación es independiente del lenguaje de programación utilizado a pesar que fue diseñada para JavaScript. Es muy simple de usar y de generar una documentación publicable en internet. La misma se puede observar en la figura 1.

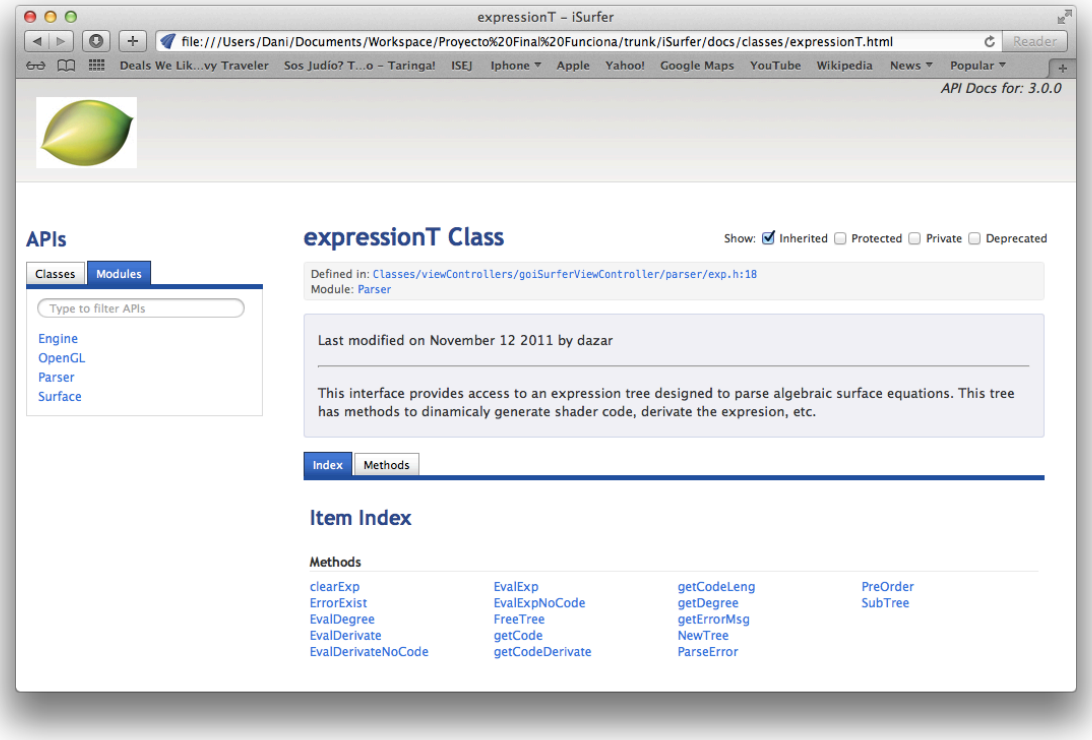


Figura 1. Documentación del código utilizando YuiDoc

iSurfer

Introducción

La idea original del proyecto es crear una aplicación para celulares para graficar superficies algebraicas. Se eligió como plataforma a los dispositivos que cuentan con iOS como sistema operativo. Para graficar las superficies existen varios métodos y algoritmos, los cuales difieren en performance y calidad de la imagen generada. Dicho graficador tiene que ser lo mas fiel posible a la matemática por lo que se descartan los algoritmos que utilizan teselado de la superficie. Debido al poco nivel de procesamiento del CPU de estos equipos se procedió a realizar un mix entre CPU y GPU para el procesamiento.

Para la implementación se utilizo OpenGL ES 2.0 para crear shaders los cuales se ejecutan en la GPU. Estos shaders se compilan en tiempo de ejecución para cada ecuación y están conformados por una parte estática y un código que genera el Parser de las formulas. Dicho Parser de las formulas matemáticas de las superficies genera código de OpenGL a partir de la entrada de una formula. La salida de los shaders de OpenGL generan las imágenes de las superficies con un algoritmo que se asemeja al de un RayTracer.

Esquema de la Aplicación

La aplicación se puede dividir en tres grandes partes aplicando el patrón MVC:

El modelo esta conformado por el Parser de ecuaciones y los Shader de OpenGL. El Controlador esta conformado por distintos objetos que controlan el Shader de OpenGL entre estos se pueden destacar los que manejan la rotación y el zoom de las superficies. La View se diseño utilizando las librerías estándar de iOS.

Para el desarrollo de la aplicación se utilizaron varios lenguajes de programación distintos

- ANSI C
- C++
- OpenGL ES 2.0
- Objetive C
- HTML

De decidió utilizar estos lenguajes de programación de forma tal que permitan migrar la aplicación fácilmente a otras plataformas como pueden ser Android o Windows 8. Para el Shader se utilizo OpenGL ES 2.0 debido a que todavía no existen librerías disponibles para utilizar OpenCL y realizar una integración mas simple de GPU y CPU. A partir de OpenGL ES 2.0 se pueden escribir shaders los cuales permiten utilizar la GPU para el procesamiento de datos. Para el parser y el manejo del GPU se utilizo una mezcla entre ANSI C y C++. Esto se decidió de forma de generar un código fuente estándar que se pueda reutilizar para las distintas plataformas. Objective C solo fue utilizado para el manejo de la interfaz grafica.

El programa se puede dividir en 5 grandes partes:

1. El Parser.
2. La Interfaz.
3. Los Shaders.
4. El Engine.
5. Las Galerías.

El Parser

Esta escrito en C. Para esto se utilizo código del libro “Programming Abstractions in C: A Second Course in Computer Science”. Armando un árbol de expresión donde en cada nodo hoja hay un numero o una variable y cada nodo interno es un operador ver figura 2. Cuando viene un par de paréntesis se balancea el árbol para seguir manteniendo la precedencia matemática.

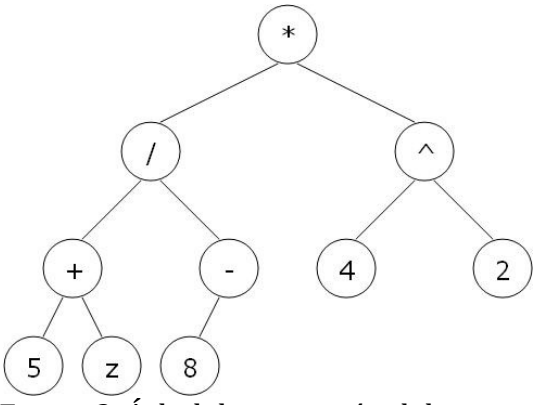


Figura 2. Árbol de expresión del parser.

Una vez generado el árbol de expresiones el mismo cuenta con funciones para calcular el grado del polinomio. La salida del parser es el código de OpenGL necesario para parametrizar el polinomio en X(t), Y(t) y Z(t) y calcular las derivadas parciales para obtener la normal de la superficie algebraica.

La Interfaz

La interfaz del usuario paso por varias transformaciones a lo largo del proyecto. Inicialmente se planteo un diseño inicial como se puede ver en la figura 3. El mismo fue cambiando al agregar distintas funcionalidades. Hasta que una vez que todo el engine funcionaba correctamente se decidió hacer un refactor del mismo por una interfaz mas simple. Para esto se utilizaron ideas y diseños aportados por un diseñador en Alemania llamado Christoph Knoth. Dicha interfaz se puede ver en la Figura 4.

Para mas información referirse a la carpeta de diseño adjunta, la cual cuenta con la documentación de las mismas.

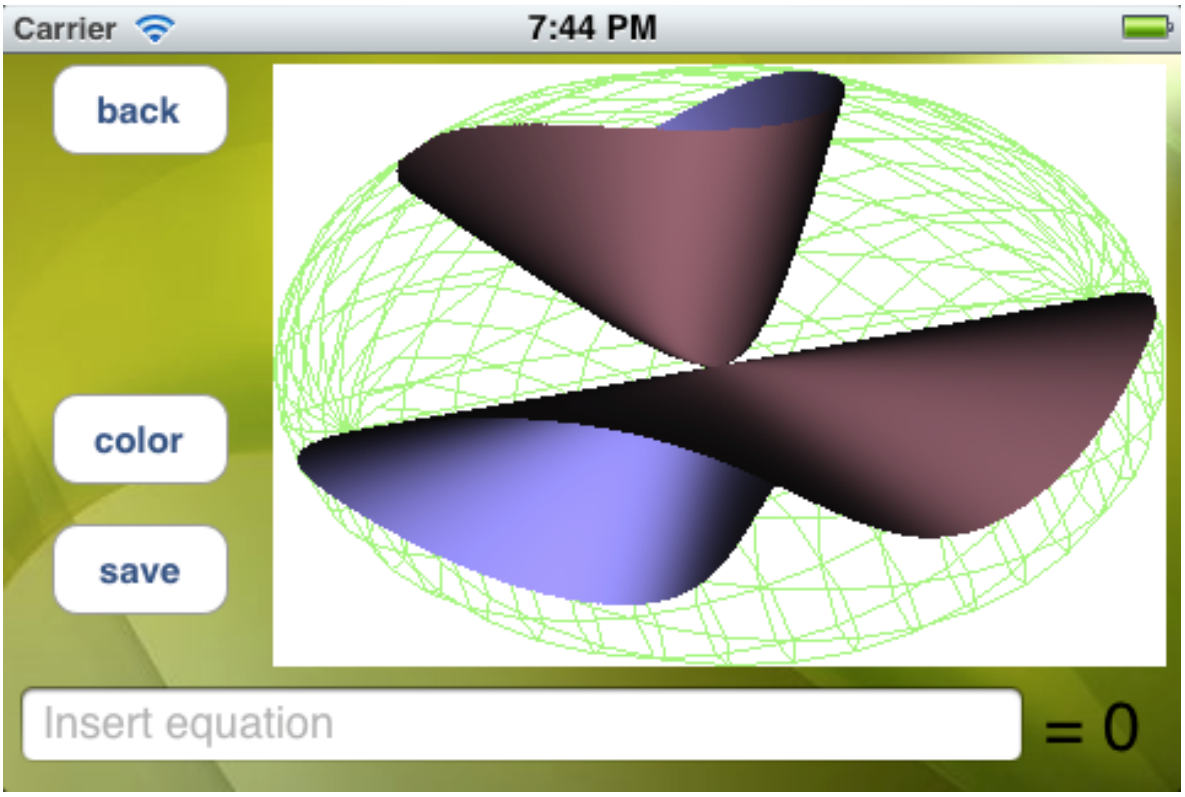


Figura 3. Diseño inicial de la aplicación.

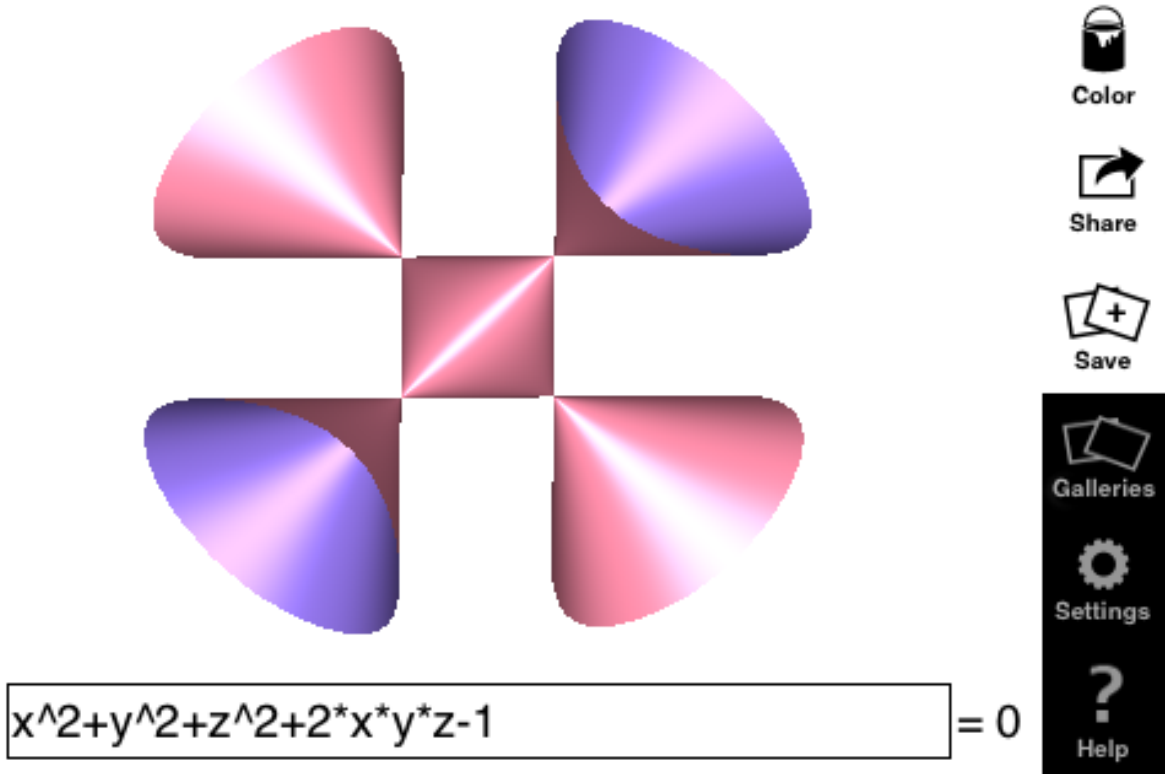


Figura 4 Diseño final

La Base de Datos

El motor de base de datos de la aplicación es SQLite que es el estándar en aplicaciones para dispositivos móviles. En la misma se guardan las galerías, superficies. Tanto las creadas por el usuario como las predeterminadas. Cada Galería cuenta con una colección de superficies. Todas las descripciones, títulos, etc. Se encuentran internacionalizadas en ingles y español.

Las Galerías

Las galerías son una parte fundamental de la aplicación ya que son el lugar donde el usuario puede aprender mas sobre la matemática inherente a las superficies algebraicas ver figuras 5, 6 y 7. El usuario puede elegir una superficie de la galería y comenzar a usar la aplicación.

Existen dos tipos de galerías distintas:

- De ejemplo que contienen bastas explicaciones sobre la superficie o método matemático en cuestión y imágenes del mundo real similares. Estas galerías no pueden ser borradas ya que conforman el manual de uso de la aplicación.
- Creadas por el usuario, las cuales pueden ser borradas y editadas. En estas galerías el usuario puede guardar cualquier superficie que encuentra para luego poder editarla.

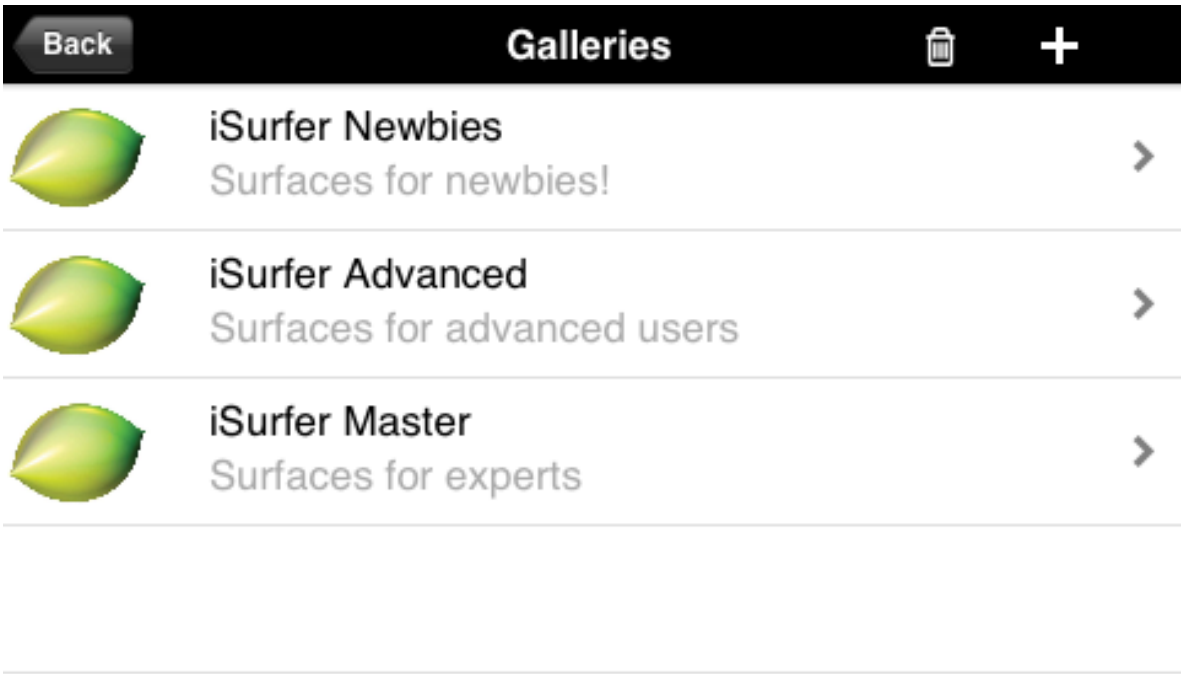


Figura 5. Listado de Galerías.

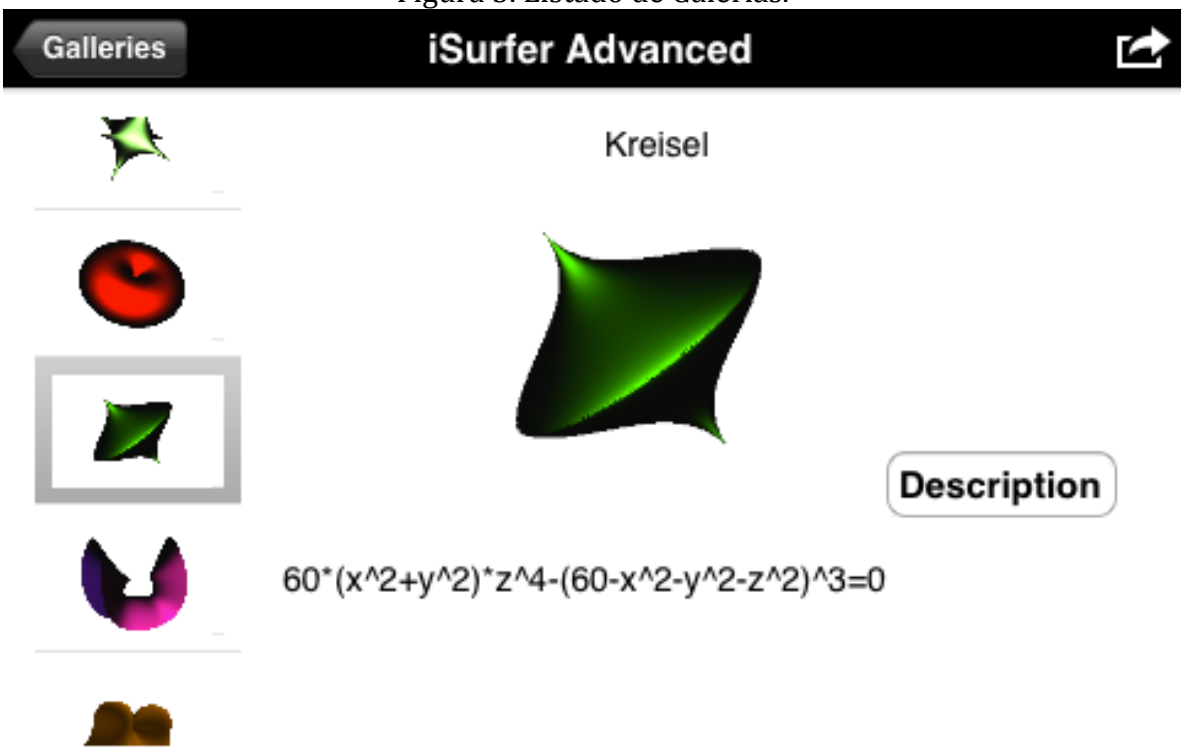


Figura 6. Detalle de una superficie en una galería.

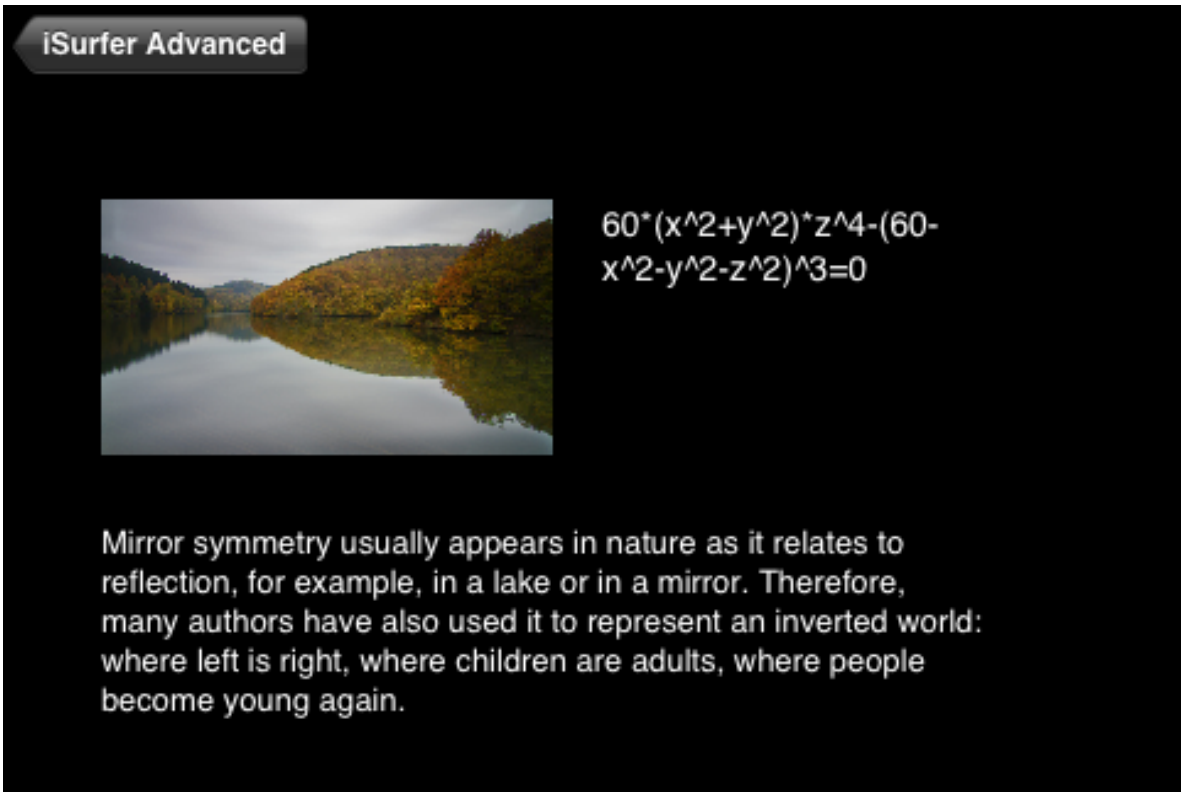


Figura 7. Descripción avanzada de la matemática inherente.

El Engine

Se diseño un conjunto de clases en C++ que son los encargados de realizar la interacción con OpenGL. Para esta sección nos basamos en el libro "*iPhone 3D Programming*". Dentro de estas clases se pueden distinguir dos secciones principalmente el Engine que es genérico para cualquier aplicación que quiera usar OpenGL ES 2.0 y Surface que es lo que contiene todas las variables y matrices de transformaciones que requerimos pasar a los Shaders.

El Engine esta conformado por clases que inicializan OpenGL. Librerías matemáticas necesarias, manejo de matrices, quaterniones y vectores. Permiten generar algunos objetos por tesselasión como puede ser una esfera. Estos objetos conforman las Bounding Box del Ray Tracer. En un momento del desarrollo se decidió agregar varios tipos de Bounding Box distintas, pero por una cuestión de que excepto por la esfera y la caja el resto puede dificultar entender la superficie calculada se procedió a dejarlos fuera de la primer versión de la aplicación. En el Anexo Superficies de corte se pueden ver 6 tipos de superficies de corte distintas.

El paquete surface contiene las clases que permiten cambiar colores, luces, tipos de camera, etc. Es el encargado de generar todas las matrices de Transformación

que son requeridas por el programa. Es el encargado de compilar los Shaders, esto implica tomar el código estático juntarlo con la ecuación, su derivada y todo lo que dependa de la ecuación. Luego se compila y se genera una programa ejecutable dentro de la GPU.

Los Shaders

Existen dos programas distintos dentro de OpenGL. Uno que muestra los triángulos de la Bounding Box con color verde. El otro es el que se encarga de hacer todo lo necesario para mostrar las superficies algebraicas.

El programa principal esta compuesto por el Vertex y el Fragment Shader. El Vertex Shader recibe la matriz del modelo y la de proyección. La del modelo esta compuesta por la matriz de escala, la de translación y la de rotación. Las primeras dos se modifican dependiendo del zoom mientras que la de rotación es la única matriz que se modifica al rotar. La de proyección tiene la cámara, la cual también se modifica al hacer zoom. También es necesario pasar la posición de cada triangulo que va a procesar OpenGL. En base a estos OpenGL nos generara los rayos y invocara al fragment shader si y solo si el rayo se intersecta dentro del campo de visión de la cámara. Pero como a nosotros nos interesa la dirección del rayo y su origen hacemos un calculo inverso por cada rayo para obtenerlos y pasárselo al fragment shader. Una cuestión a notar es que puede que por cada triangulo se invoquen varias veces al fragment shader con los distintos valores aproximando la diferencia entre los vértices de los triángulos.

El Fragment Shader debería ser llamado una vez por cada pixel, pero puede que sea llamado mas de una vez. Por esta razón debe estar lo mas optimizado posible. En un principio obtenemos el Tmin y Tmax según una esfera contenedora para limitar el RayTracing. Una vez realizado eso buscamos las raíces dependiendo del grado de la superficie. El algoritmo a utilizar se modifica en tiempo de compilación dependiendo del grado del polinomio. Una vez obtenido el punto donde hay una raíz si es que existe utilizamos ese punto para calcular las luces. Para el calculo de luces optimizado utilizamos la normal de la superficie en el punto de intersección, esta se obtiene calculando el valor en dicho punto de las derivadas parciales en las tres direcciones. Para esto ya no utilizamos la parametrizacion en t sino que utilizamos el valor real en el espacio de la Raíz.

Para el zoom son cuatro cosas que debemos modificar:

- El tamaño de la cámara.
- La distancia de la Bounding Box a la Cámara.
- El tamaño de la Bounding Box.

- El radio de la esfera a utilizar en el Frament Shader para obtener Tmin y Tmax.

Para la rotación se utilizo una simplificación del programa propuesta por Christian Stussak que es rotar en X e Y en función de las coordenadas que se desliza el dedo por la pantalla en X e Y. En cuanto a Z solo se rota en dicho eje si hay rotación tanto en X como en Y. Para que la rotación se pueda apreciar en tiempo real, redujimos el tamaño de la imagen resultante de OpenGL. Esto reduce el tiempo que tarda en renderizar cada frame. Luego la imagen es estirada por medio de la interfaz gráfica, lo que produce una imagen pixelada.

Una de las cuestiones mas importantes a destacar es la imposibilidad de pasar una cantidad variable de valores entre el CPU y la GPU. Por esta razón no podemos pasar los coeficientes del polinomio resultante parametrizado en t entre un frame y otro. Sino que debemos generar un nuevo shader con el polinomio parametrizado en el código fuente del Shader, compilarlo, cambiar el programa activo de la GPU y eliminar el anterior. Esto es uno de los limitantes mas grandes de la aplicación ya que la GPU no esta diseñada para esto y todo el proceso es muy lento. Una vez renderizado el primer frame de la nueva superficie la aplicación tarda unos segundos en optimizar variables para funcionar velozmente.

Durante el desarrollo se aplicaron distintas técnicas sobre el render de las superficies como puede ser el caso de Toon Shader, Texturas, etc. En el Anexo Texturas y Toon Shader se puede observar la utilización de texturas y del toon Shader.

A futuro esto se podría mejorar una vez que se hagan publicas las Apis de OpenCL en los OS móviles.

Conclusión y Futuro

Resultados

La Aplicación se encuentra con un correcto funcionamiento. Todavía no se encuentra en el App Store de Apple por dificultades técnicas a la hora de obtener la cuenta de Developer debido al hackeo de los servidores de Apple. En cuanto a implementación se desarrollaron varias funcionalidades que están por fuera del alcance del proyecto.

Dificultades

Las principales dificultades que se nos presentaron fueron a debido a la complejidad matemática del proyecto y la poca experiencia que teníamos con OpenGL. La falta de experiencia con OpenGL se sumo a las dificultades para debuggear y encontrar errores de código por lo que perdimos mucho tiempo. Tuvimos problemas de organización durante largos periodos de tiempo. En cuanto a errores de código en un principio tuvimos varios problemas con las matrices y los cálculos matemáticos. Estos se resolvieron al cambiar las librerías matemáticas por unas mas utilizadas.

En cuanto a los algoritmos primero los probamos en C y en Matlab una vez que funcionaban los pasamos a GLSL. Esto sin embargo no nos ayudo a eliminar los errores. Primero creímos que los errores se debían a la baja precisión que tienen las variables de punto flotante en estos dispositivos. Pero al final se debían a una combinación de varios factores. Primero calculamos mal el Grado de un polinomio luego había errores en la regla de la cadena de la derivada. Una vez resuelto esto los errores continuaron.

Para poder encontrar errores dentro de un shader cuando el mismo no hace lo esperado es necesario poder conocer el estado de las variables. Debido a que este código se ejecuta en la placa de video no se puede imprimir un valor numérico de una variable. Esto da lugar al “Color Debugger” el cual se basa en imprimir los pixeles a pantalla . Si se cumple una cierta condición que nos interesa se imprime un color especifico a la pantalla en dicho pixel. El problema es que este método para encontrar errores puede llevar muchísimo tiempo y no siempre da resultados. Por eso se decidió implementar distintas partes del engine grafico en Matlab y Ansi C. De esta forma se pudo comprobar el correcto funcionamiento de cada algoritmo al analizar la entrada y la salida. Después los mismo fueron portados a GLSL.

Una vez que no se nos ocurrió que podía generar los errores nos ayudo Christian Stussak para ver que teníamos errores es la implementación de Descartes. Al reparar los mismos decidimos armar nuestra implementación de Cardano a partir de la implementación de JMonkey Engine. Una vez realizado estos cambios se solucionaron la mayoría de los artefactos raros que se presentaban pero nos quedo un ruido en la implementación de Cardano ver figura 8. Este depende de la rotación y el zoom y parece ser por la precisión del dispositivo al ser un algoritmo exacto.

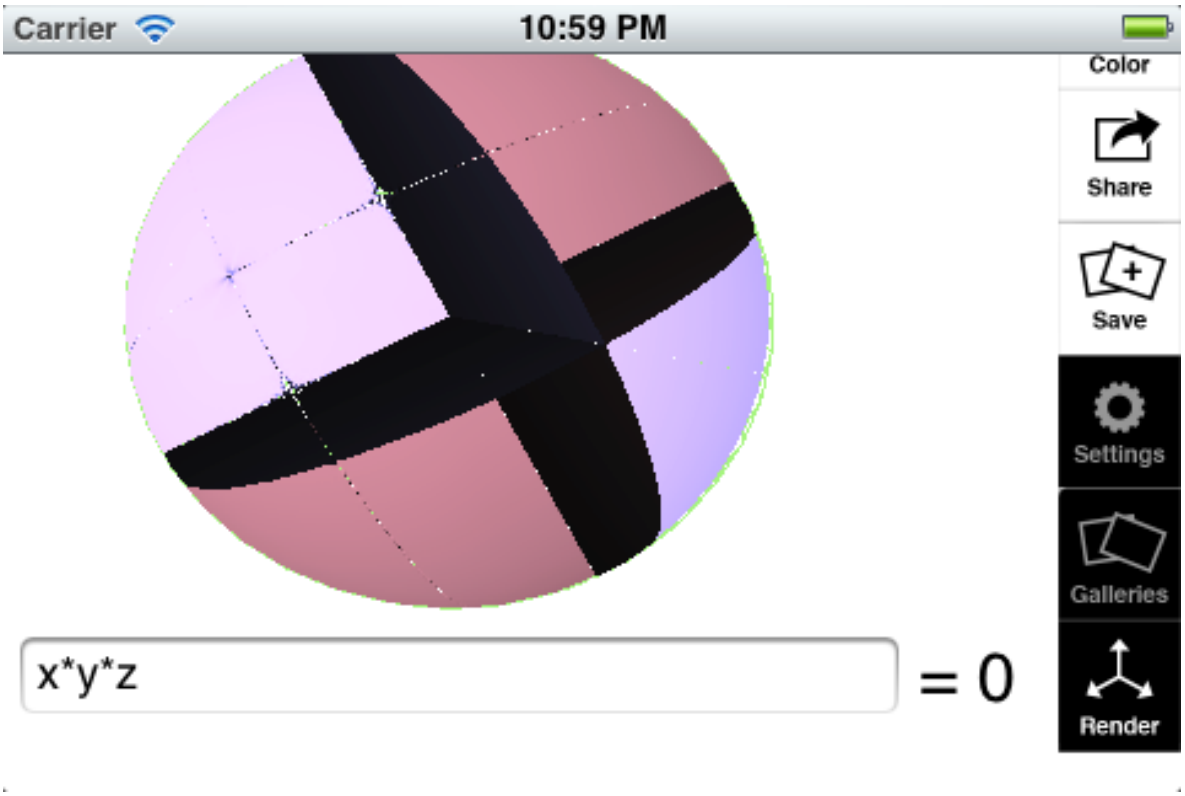


Figura 8. Contiene una cruz de ruido que gira independiente de la superficie.

También existieron varios equivocaciones menores, como puede ser tener los ejes invertidos, la rotación al revés, etc. En cuanto a la interfaz a mitad del desarrollo decidimos hacer un refactor para que la misma tuviera una interfaz mas acorde a otras aplicaciones en iOS.

Conclusiones

A lo largo de este trabajo pudimos aprender diversos temas y lenguajes de programación como Objective C y OpenGL. Si bien nos encontramos con diversos problemas durante el desarrollo del mismo, pudimos finalmente sortearlos con perseverancia y trabajo de equipo.

Uno de los mayores desafíos fue dominar la técnica del raytracing y la implementación en OpenGL 2. La tarea se nos dificultó aún más al no contar con alguien en el país que dominara este lenguaje y tuviera la base matemática necesaria.

A pesar de todas las restricciones a nivel software y hardware de estos dispositivos y los benchmarks iniciales desalentadores, pudimos llegar a hacer una implementación razonablemente usable.

Futuro

Existen varias cosas que se pueden agregar a la aplicación:

- Una Versión de iPad con una interfaz optimizada para tabletas.
- Una Versión para Android y otras plataformas que soporten OpenGL ES 2.0.
- Utilizar una web de gestión de galerías compatible en común con la versión Desktop de Surfer y integración con redes sociales.
- Agregar distintos algoritmos seleccionables para el renderizado.
- A futuro se podría investigar la utilización de OpenCL que es un lenguaje que esta surgiendo para el calculo combinado sobre CPU y GPU. Dicho lenguaje no fue utilizado debido que por ahora no esta disponible en dispositivos móviles.
- Otra posible mejora seria tener un shader compilado para cada grado / algoritmo y pasarle por parámetros los valores de los coeficientes del polinomio.

Anexo

Manual de instalación.

Para instalar la aplicación es necesario bajar la misma del AppStore de Apple como cualquier otra aplicación de iOS.

Paquete de instalación del software en producción

Para instalar el paquete del software en producción es necesario primero instalar la IDE de desarrollo. Es decir inicialmente se debe descargar y instalar el XCode, al momento de armar este documento la versión 4.6. Luego se debe abrir el archivo iSurfer.xcodeproj localizado en el repositorio. Por ultimo es necesario seleccionar si se va a ejecutar / compilar el código para el simulador o para un dispositivo conectado a la PC ver Figura 9. Para poder hacer un deploy a un dispositivo se debe contar con las claves generadas por el mismo al tener una cuenta de Desarrollador de Apple.

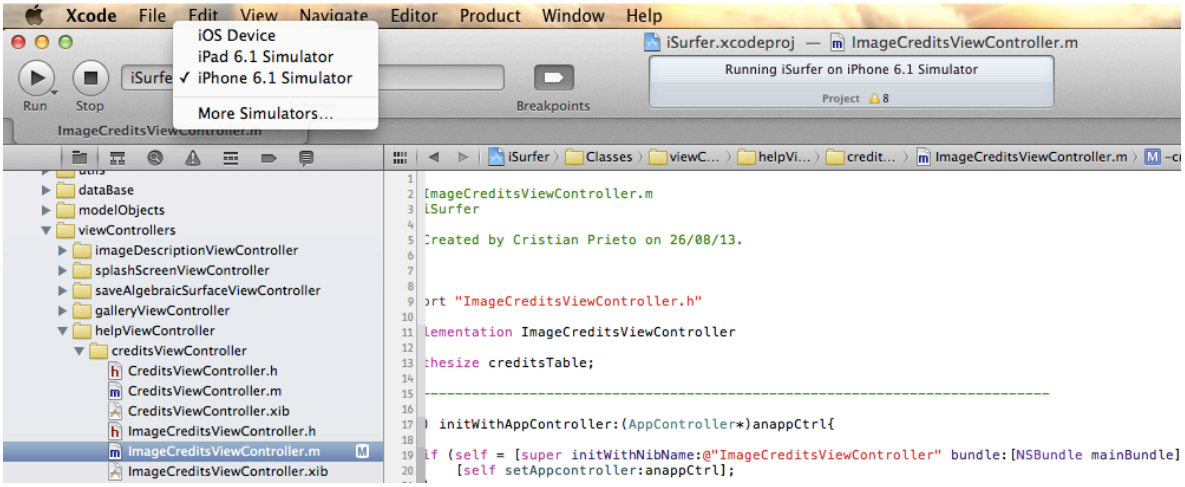


Figura 9. Opcion para elegir el dispositivo a ejecutar la App en XCode.

Superficies de corte

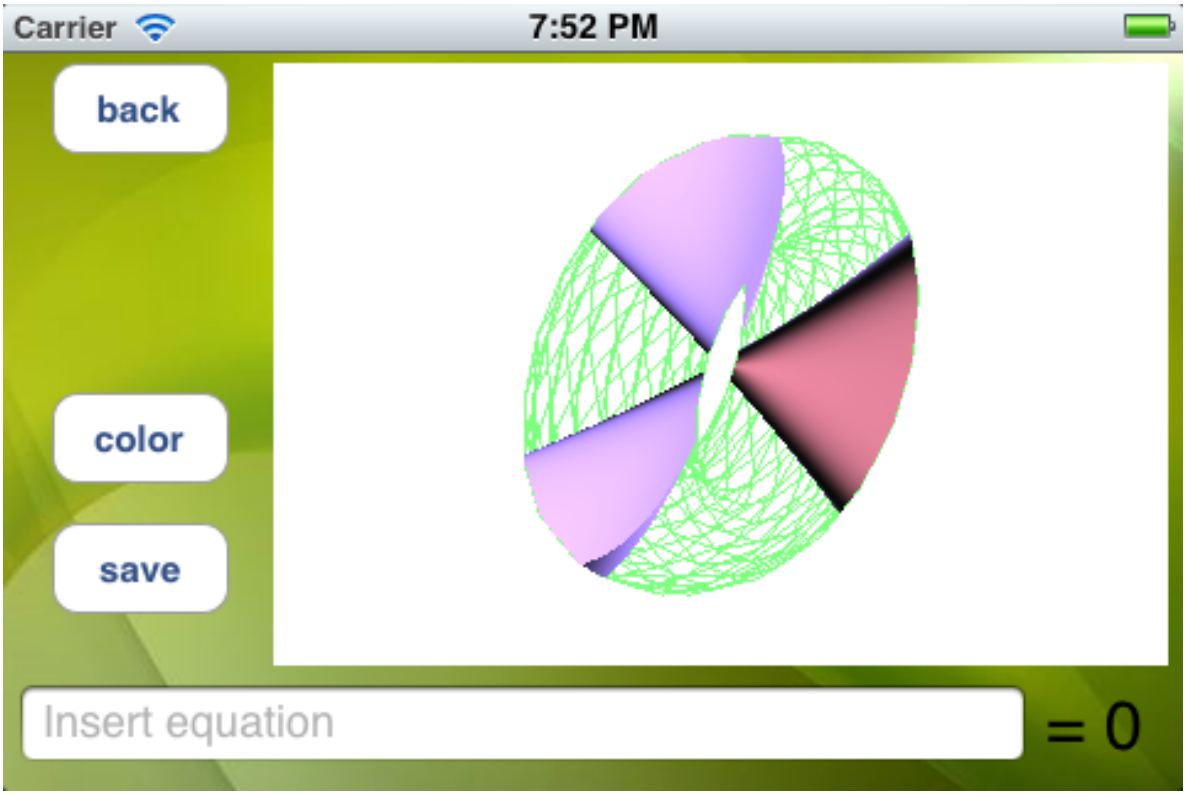


Figura 10. Torus como BB.

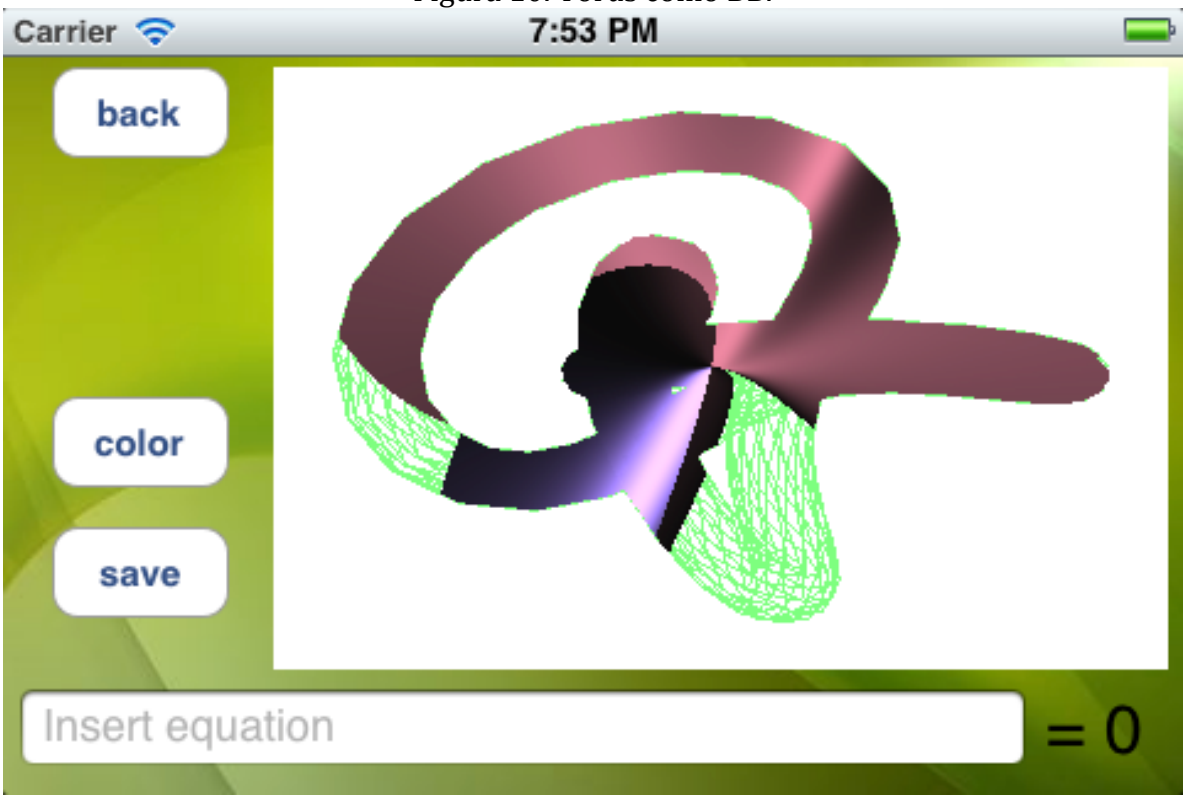


Figura 11. Knot como BB.

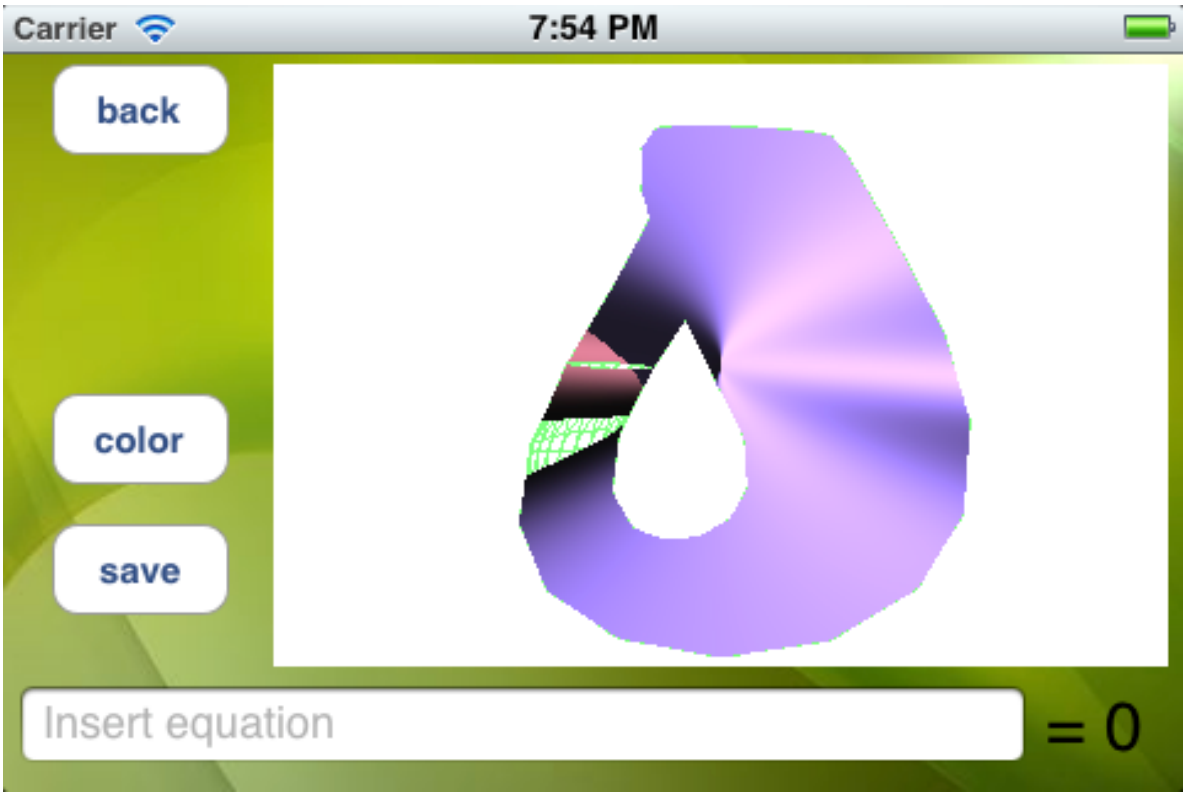


Figura 12. Klein Bottle como BB.

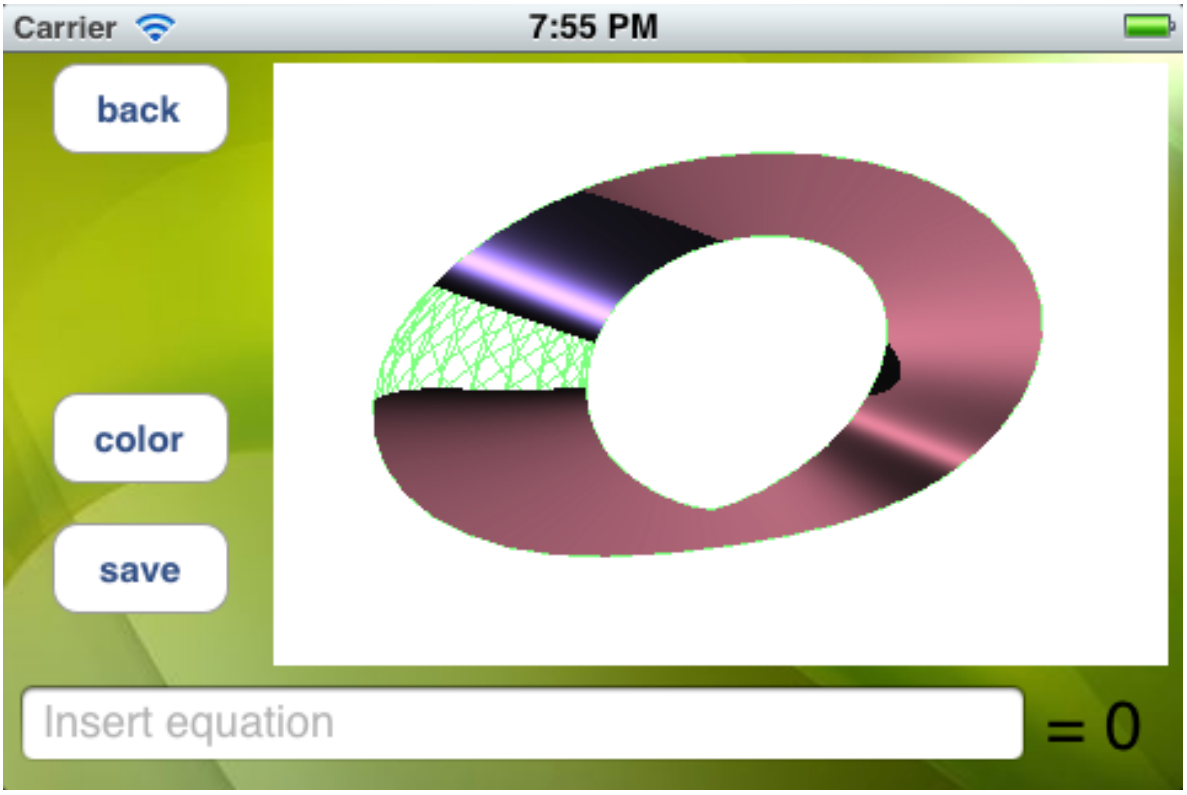


Figura 13. Cinta de moebius como BB.

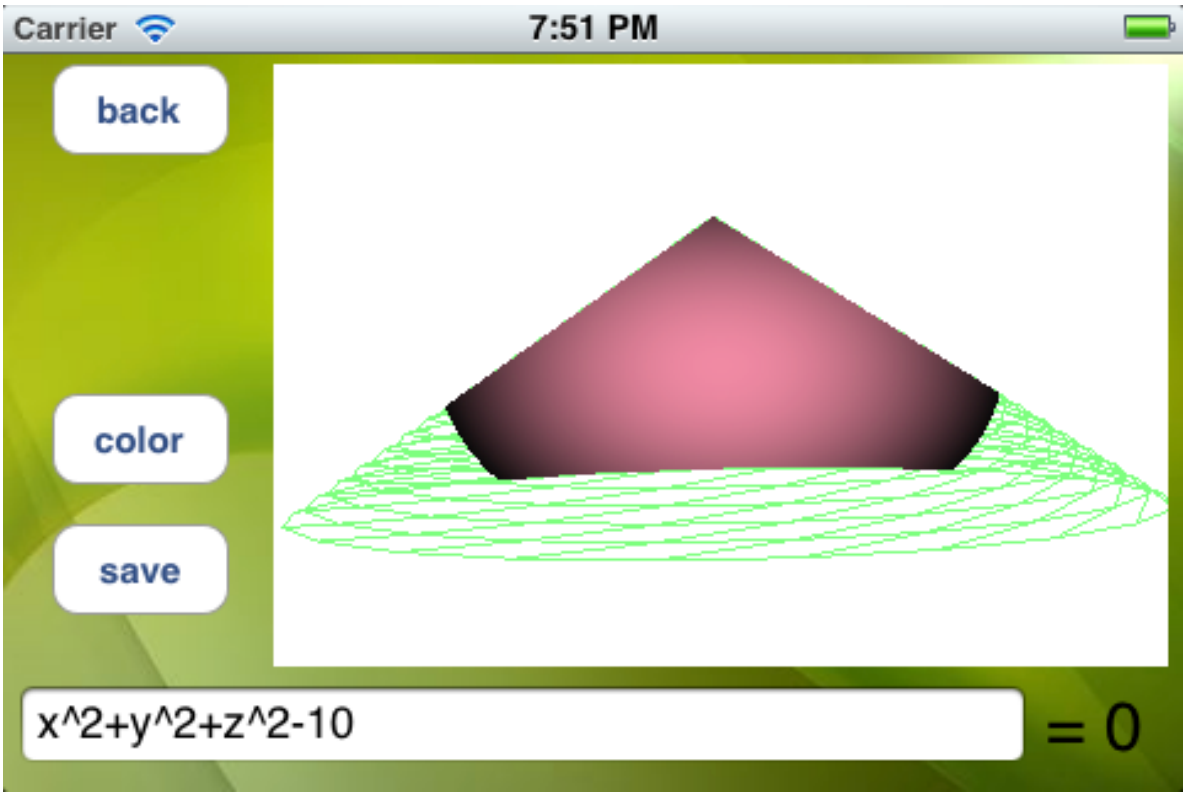


Figura 14. Cono como BB.

Texturas y Toon Shader

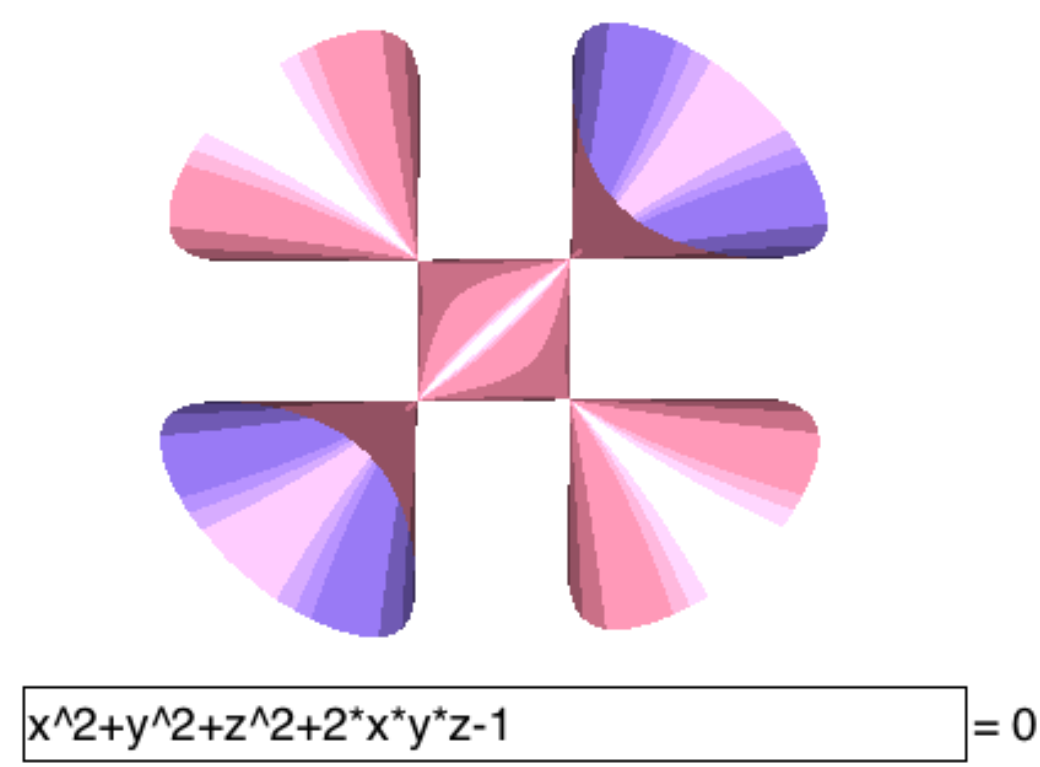


Figura 15. Efecto del Cell Shader.

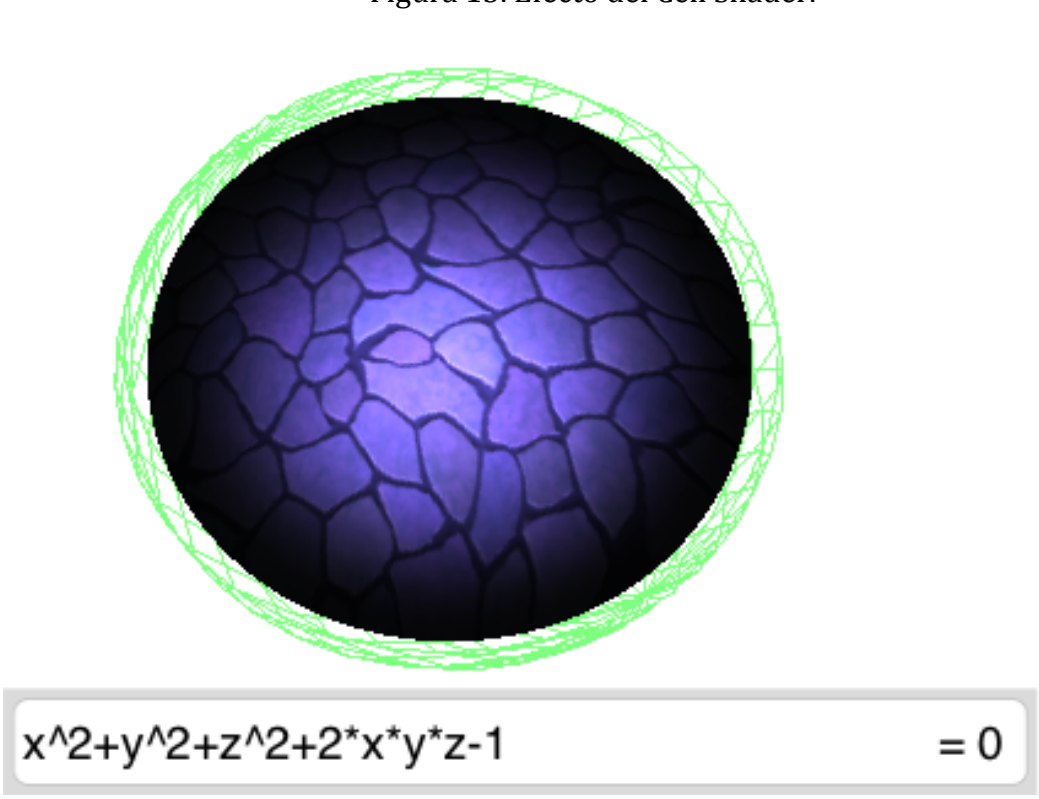


figura 16. Textura de Mármol Sobre una esfera.

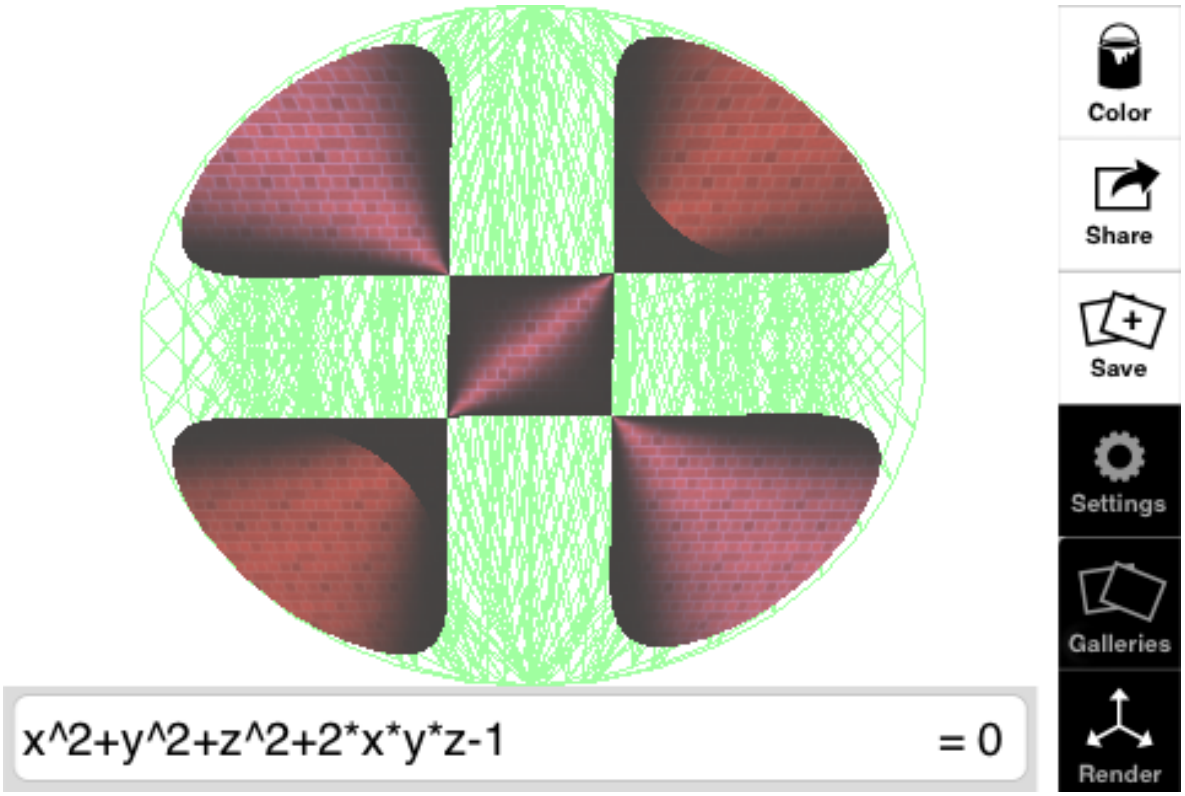
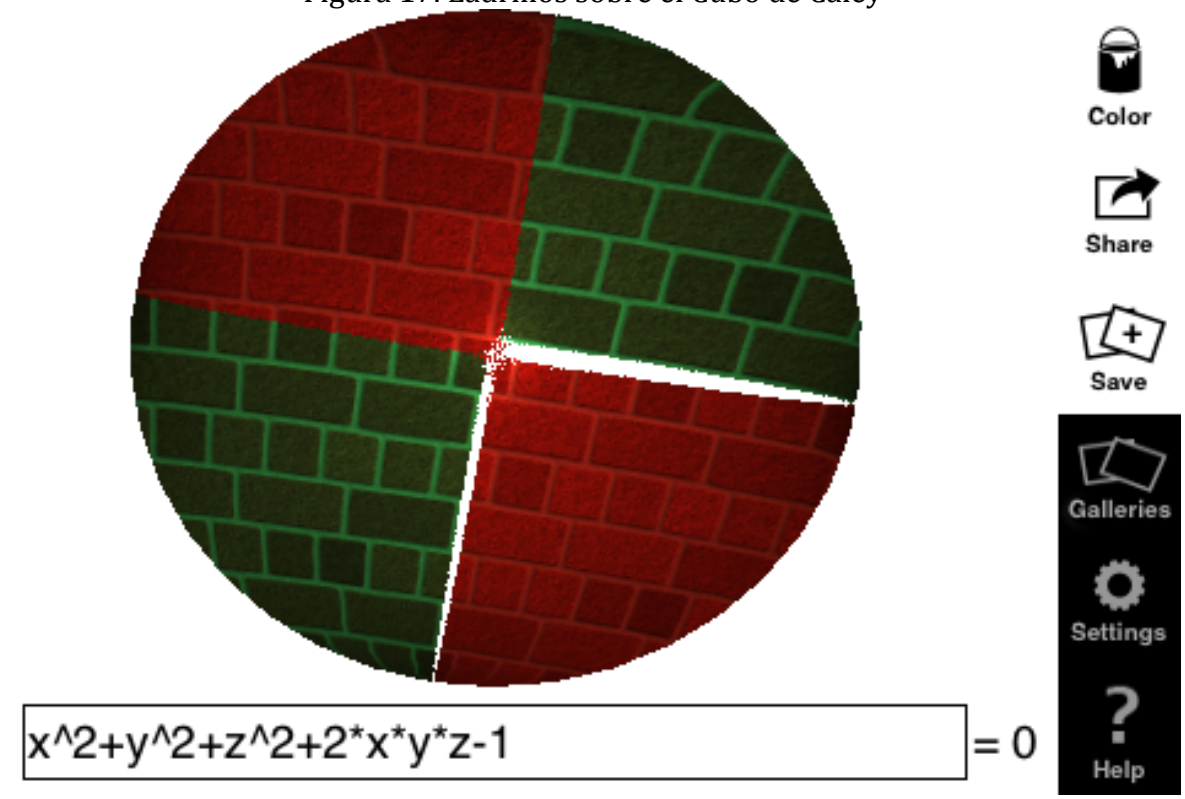


Figura 17. Ladrillos sobre el Cubo de Caley



Figra 18. Muestra el uso de Texturas y Colores sobre las texturas. También se puede aplicar el efecto de Toon shader sobre las texturas

Referencias

Programming Abstractions in C: A Second Course in Computer Science

JMonkey Engine

iPhone 3D Programming O'Reilly

http://www.khronos.org/opengles/2_X/

Ch. Stussak. *Echtzeit-Raytracing algebraischer Flächen auf der Graphics Processing Unit*. Diplomarbeit. 2007.

Algebraic Surfaces Website.
http://www.algebraicsurface.net.

G. Barczik, O. Labs, and D. Lordick. Algebraic geometry in architectural
design.

E. Faber and H. Hauser. Todays menu: Geometry and resolution of singular algebraic surfaces. AMERICAN MATHEMATICAL SOCIETY, 47(3):373– 417, 2010.

A.S. Glassner. An introduction to ray tracing. Morgan Kaufmann, 1989.

G.-M. Greuel and A.D. Matt. IMAGINARY through the eyes of mathe-
matics. Travelling Exhibition Catalogue. 2009.

IMAGINARY.
http://www.imaginary-exhibition.com.

Introduction to Algebraic Geometry.
http://www.math.purdue.edu/~dvb/algeom.html.

O. Labs. A list of challenges for real algebraic plane curve visualization software. Nonlinear Computational Geometry, pages 137–164, 2010.

A. Matt. IMAGINARY and the idea of an open source math exhibition platform. Raising Public Mathematical Awareness, Springer Verlag, 2011.

SURFER - visualization of algebraic surfaces. http://www.imaginary-exhibition.com/surfer.

Oberwolfach, Yesterday and Today, Jackson 2000