Lucía Plaza Pérez Pablo Rodríguez Fernández Iban Serrano Caballero

Introducción

HOPENGL

ÍNDICE

1

INTRODUCCIÓN

INTRODUCCIÓN

- Los programas de visualización necesitan un conjunto de herramientas gráficas 2D y 3D cercanas al hardware

- HOpenGL: es un conjunto de librerías que permiten utilizar OpenGL desde el lenguaje Haskell.

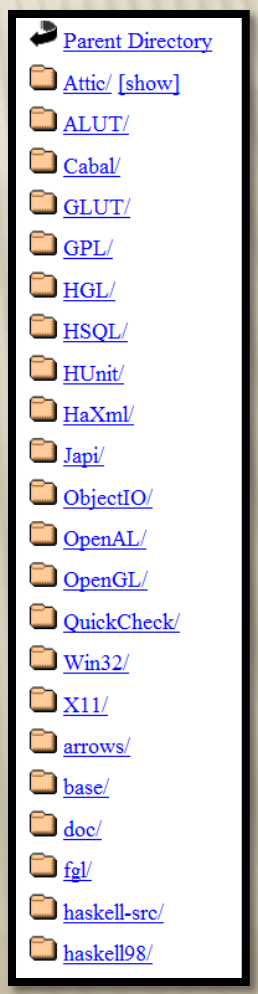
INTRODUCCIÓN

- Sven Panne es el responsable de HOpenGL.
- HOpenGL es una abreviatura de Haskell Open Graphics Library.
- Actualmente se utiliza como una herramienta vinculante más que como una librería.
- Entre otras características: proporciona un tipificado fuerte y un interfaz para OpenGL más al estilo de Haskell.

INTRODUCCIÓN

- Al comienzo, las fuentes de HOpenGl residían en su propio repositorio, lo cual producía constantes problemas de compatibilidad.

- Las fuentes han sido movidas a la parte de librerías jerárquicas del repositorio de fptools.



INTRODUCCIÓN

- VENTAJAS de HOpenGL:
  - Es una herramienta muy útil a la hora de crear animaciones gráficas o desarrollar juegos, debido a su elasticidad a la hora de trabajar con gráficos.
  - Facilidad a la hora de trabajar con gráficos, con unas 10 instrucciones podremos ver ante nosotros una impresionante obra de arte.

INTRODUCCIÓN

- INCONVENIENTES de HOpenGL:
  - La instalación no es sencilla, debido a que necesita muchos requisitos y dependiendo de la versión que se instale se tendrán diferentes problemas.
  - Dependencia con la tarjeta gráfica: dependiendo de la tarjeta gráfica muestra el resultado esperado o no.

2

HASKELL, OPENGL Y HOPENGL

HASKELL, OPENGL Y HOPENGL

- **Haskell:** es un lenguaje de programación puramente funcional de propósito general y fuertemente tipificado. (Haskell Curry) (1990)

  - Implementaciones:
    - *Hugs* es un intérprete. Ofrece una compilación rápida de los programas y un tiempo razonable de ejecución.
    - *GHC* "Glasgow Haskell Compiler"
      - Compila a código nativo en una variedad de arquitecturas y puede también compilar a C.
      - Tiene unas cuantas librerías (por ejemplo OpenGL) que, aunque muy útiles, sólo funcionan bajo GHC.

HASKELL, OPENGL Y HOPENGL

- **OpenGL:** es una librería estandarizada para la descripción de escenas tridimensionales.
  - Es independiente del sistema operativo.
  - Se encuentra disponible para distintas plataformas.
  - La librería es bastante completa y eficiente.
  - Es una de las librerías 3D más utilizadas actualmente.

HASKELL, OPENGL Y HOPENGL

- OpenGL: Fue desarrollada originalmente por Silicon Graphics Inc. (**SGI**)(*) en 1992
- se usa ampliamente en
  - realidad virtual
  - visualización de información
  - simulación de vuelo.
  - desarrollo de videojuegos, donde compite con Direct3D en plataformas Microsoft Windows.

(*)SGI Computación de alto rendimiento, visualización y almacenamiento. Fue fundada por Jim Clark y Abbey Silverstone en 1982.

HASKELL, OPENGL Y HOPENGL

- Los soportes de HOpenGL para Haskell utilizan la biblioteca de los gráficos de OpenGL y sus herramientas, como son Glut, Glu y Gl.

- Las herramientas que permiten entre otras cosas *la gestión de periféricos*

  => deberemos tener *instalado* en nuestra maquina de trabajo el *OpenGL.*

HASKELL, OPENGL Y HOPENGL

- Bibliotecas externas que añaden características no disponibles en el propio OpenGL:

  * GLU: Ofrece funciones de dibujo de alto nivel basadas en primitivas de OpenGL.

  * GLUT: API multiplataforma que facilita el manejo de ventanas e interacción por medio de teclado y ratón.

  * GLUI: Interfaz de usuario basada en GLUT; proporciona elementos de control tales como botones, cajas de selección..

HASKELL, OPENGL Y HOPENGL

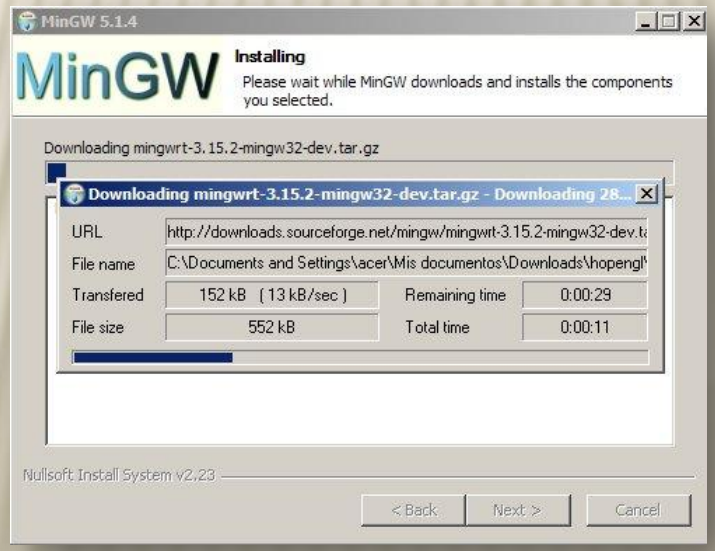
- Utilización en la universidad
  - OpenGL
    - Curso de OpenGL en la UNED
    - Asignatura “Computación Gráfica” en la Escuela Universitaria de Enfermería de Santiago de Compostela.
      - https://www.usc.es/es/centros/enfermaria_stgo/materia.jsp?materia=30026&ano=59&idioma=2
    - La asignatura de Informática Industrial de la Escuela Universitaria de Ingeniería Técnica Industrial (E.U.I.T.I) de la Universidad Politécnica de Madrid.
      - http://taee2008.unizar.es/papers/p52.pdf
  - HOpenGL
    - Proyecto fin de carrera E.T.I.S (2006) “Una librería para animaciones funcionales reactivas en 3D con Haskell y OpenGL” Director: Jose Enrique Gallardo Ruiz.

3

INSTALACIÓN DEL ENTORNO

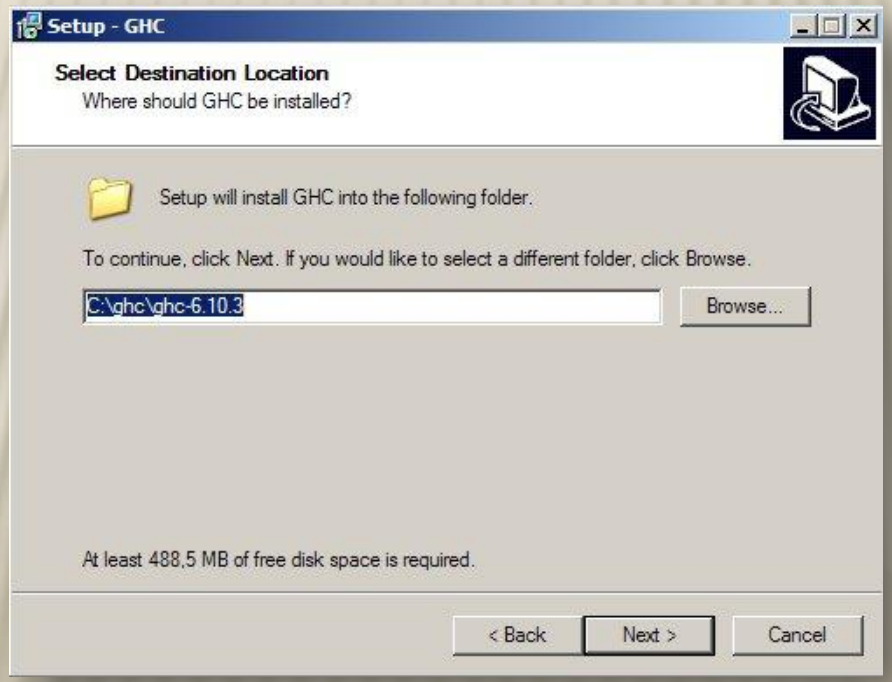
WINDOWS

- MinGW y MSYS: GNU para Windows y utilidades como ‘make’, ‘grep’ o ‘gcc’.
  - + MinGW Debe instalarse la instalación mínima.
  - + MSYS Selecciona “post instalación” e indica el directorio de instalación de MinGW.



WINDOWS

- GHC (Glasgow Haskell Compiler)
- Descarga e instala la versión del binario para Windows



WINDOWS

- Compilar e instalar freeglut
  - Descarga freeglut 2.4.0 source distribution y Descomprimelo en tu directorio home de msys que por defecto es:

    C:\msys\1.0\home\<Tu nombre deusuario>
  - Abre una sesión de msys y escribe los siguientes comandos:

```
cd freeglut-2.4.0/src/
gcc -O2 -c -DFREEGLUT_EXPORTS *.c -I../include
gcc -shared -o glut32.dll *.o -Wl,--enable-stdcall-fixup,--out-
    implib,libglut32.a -lopengl32 -lglu32 -lgdi32 –lwinmm
cp glut32.dll /c/WINDOWS/system32
cp libglut32.a /c/ghc/ghc-6.10.3/gcc-lib
```

WINDOWS

- Haskell OpenGL Binding
  - Descarga Haskell OpenGL Binding
  - Descomprímelo en tu directorio home de MSYS
  - Escribe los siguientes comandos en MSYS

```
cd OpenGL-2.2.2.0
runhaskell Setup configure
runhaskell Setup build
runhaskell Setup install
```

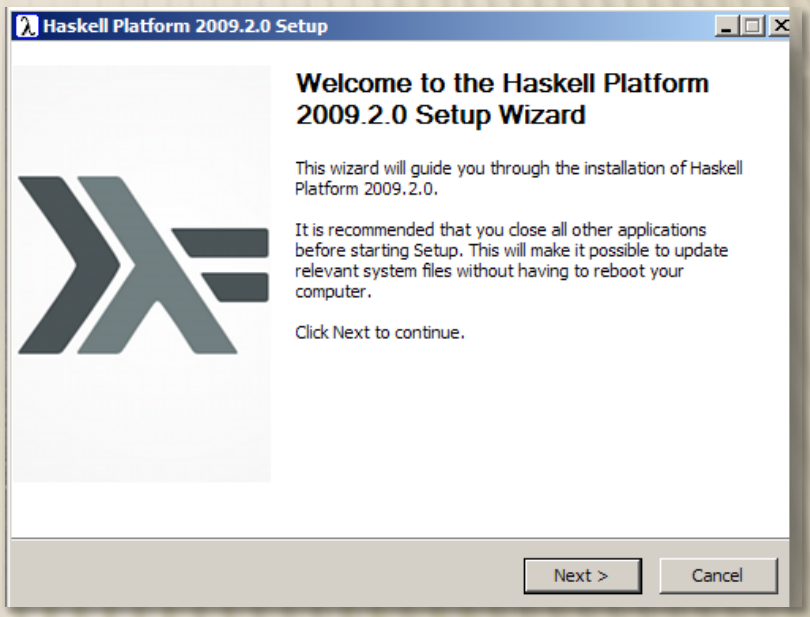
WINDOWS

- Haskell GLUT Binding
  - Descarga el Haskell GLUT Binding
  - Descarga el patch glutwin2112
  - Descomprímelos en tu directorio home de MSYS
  - Escribe los siguientes comandos en MSYS

```
cd GLUT-2.1.2.1
patch -p1 < ../glutWin2112.patch
C_INCLUDE_PATH="../freeglut-2.4.0/include/“
    LIBRARY_PATH="../freeglut-2.4.0/src/" runhaskell Setup configure
C_INCLUDE_PATH="../freeglut-2.4.0/include/"
    LIBRARY_PATH="../freeglut-2.4.0/src/" runhaskell Setup build
C_INCLUDE_PATH="../freeglut-2.4.0/include/"
    LIBRARY_PATH="../freeglut-2.4.0/src/" runhaskell Setup install
```

WINDOWS (2)

- Recientemente se ha creado Haskell Platform para facilitar a los usuarios de todas las plataformas la programación en Haskell.
- Debemos tener instaladas las DLL de OpenGL GLUT (Descargar de la página oficial de OpenGL y copiarlas en Windows/System32)
- Después instalamos Haskell Platform.



UBUNTU

- Instala las librerías GLUT para GHC
  - Abre una consola y escribe el siguiente comando:

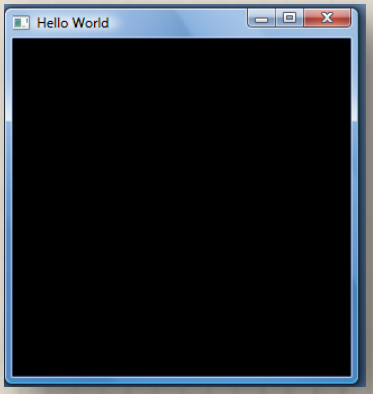
```
sudo apt-get libghc6-glut-dev
```

  - Automaticamente se instalarán todas las dependencias, incluyendo el compilador GHC y las librerías HopenGL

4

CREACIÓN DE UN PROGRAMA HOPENGL

CREACIÓN DE UN PROGRAMA HOPENGL



- Para empezar, creamos una ventana:

```
import Graphics.Rendering.OpenGL
import Graphics.UI.GLUT
main = do
        (progname, _) <- getArgsAndInitialize
        createWindow "Hello World“
        displayCallback $= clear [ ColorBuffer ]
        mainLoop
```

Limpio la pantalla

CREACIÓN DE UN PROGRAMA HOPENGL

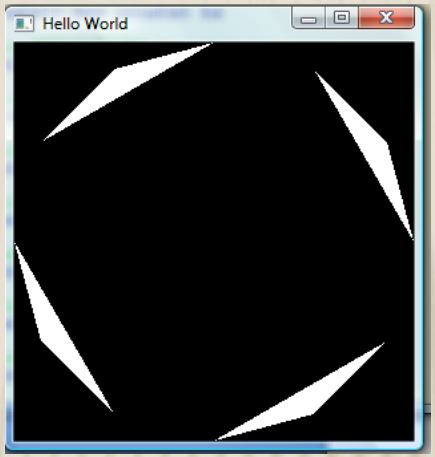
- El operador $= viene definido como:

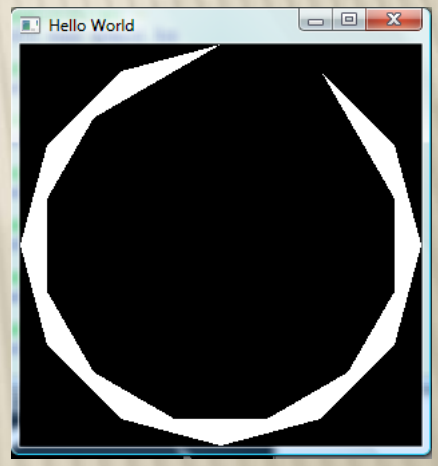
```
Infixr 2 $=
class HasSetter s where
($=) :: s a -> a -> IO
```

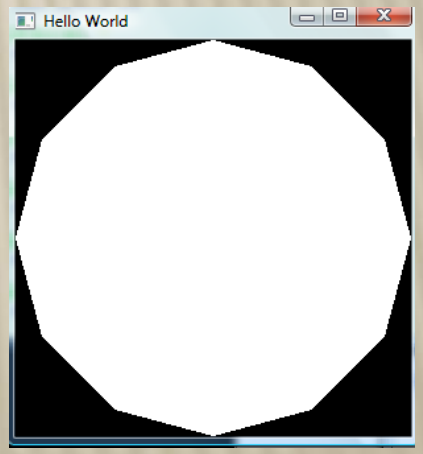
- Si tenemos a1 $= a2 , $= actúa de dos formas:
  - Si a1 es IORef, asigna el valor de a2 en a1.
  - Si a1 es algún tipo de variable de estado de OpenGL, cambiará la parte correspondiente en la máquina de estados de OpenGL.
- Las variables de HOpenGL que pueden cambiarse son de tipo SettableStateVar.

CREACIÓN DE UN PROGRAMA HOPENGL

- renderPrimitive:
  - 1º la figura a pintar:
    - Points
    - Polygon
    - Triangles
    - TriangleStrip
    - TriangleFan
    - Lines
    - LineLoop
    - LineStrip
    - Quads
    - QuadStrip
  - 2º los puntos a tener en cuenta:







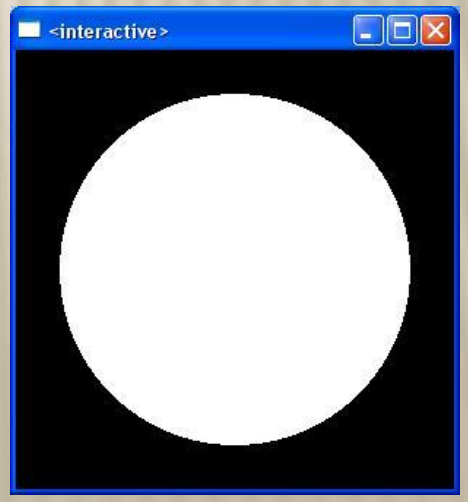

```
renderPrimitive Points $ mapM_ (\(x, y, z)->vertex$Vertex3 x y z) myPoints
Donde:
myPoints :: [(GLfloat,GLfloat,GLfloat)]
myPoints = map (\k -> (sin(2*pi*k/12),cos(2*pi*k/12),0.0)) [1..12]
```

CREACIÓN DE UN PROGRAMA HOPENGL

- Curvas y círculos:

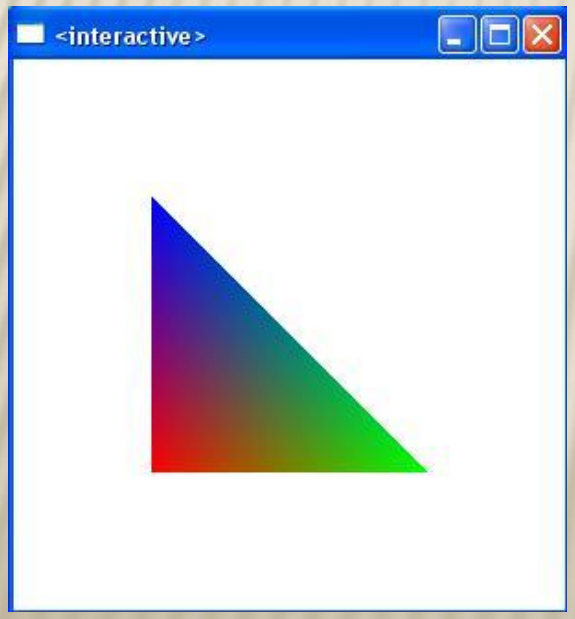
```
module Circle where

 import PointsForRendering

 import Graphics.Rendering.OpenGL
circlePoints radius number
 = [let alpha = twoPi * i /number
    in  (radius*(sin (alpha)) ,radius * (cos (alpha)),0)
   |i <- [1,2..number]]
  where
    twoPi = 2*pi
renderCircleApprox r n
 = displayPoints (circlePoints r n) LineLoop

renderCircle r = displayPoints (circle r) LineLoop
fillCircle r = displayPoints (circle r) Polygon
```

```
import PointsForRendering
import Circle

import Graphics.Rendering.OpenGL

main
 = renderInWindow $ do
     clear [ColorBuffer]
     fillCircle 0.8
```



CREACIÓN DE UN PROGRAMA HOPENGL

- Tamaño de puntos
- Atributos Linea
- Colores




```
pointSize $= 10
```

```
lineStipple $= Just (1,255)
lineWidth $= 10
```

```
renderPrimitive Triangles colorTriangle
flush
```

```
colorTriangle = do
  currentColor $= Color4 1 0 0 1
  vertex$Vertex3 (-0.5) (-0.5) (0::GLfloat)
  currentColor $= Color4 0 1 0 1
  vertex$Vertex3 (0.5) (-0.5) (0::GLfloat)
  currentColor $= Color4 0 0 1 1
  vertex$Vertex3 (-0.5) (0.5) (0::GLfloat)
```

CREACIÓN DE UN PROGRAMA HOPENGL

- A partir de aquí podemos hacer múltiples operaciones con la figura como por ejemplo:

- Traslación:

```
translate$Vector3 0.7 0.3 (0::GLfloat)
```

- Rotación:

```
rotate 30 $Vector3 0 1 (0::GLfloat)
```

- Escala:

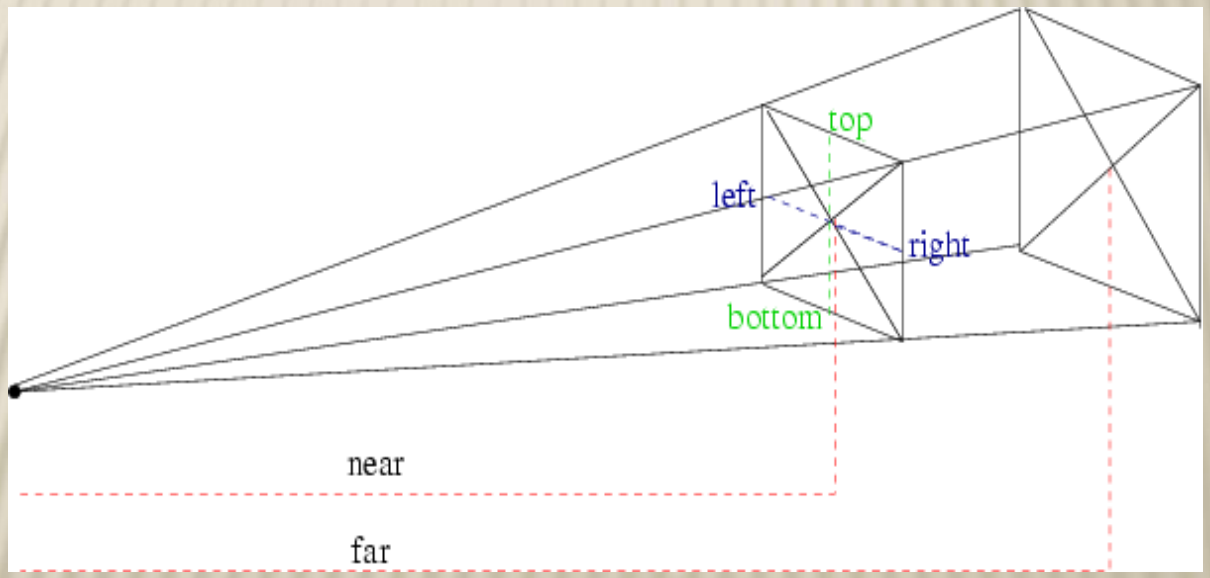
```
scale 0.3 0.9 (0.3::GLfloat)
```

CREACIÓN DE UN PROGRAMA HOPENGL

- Con la función reshape elegimos desde dónde queremos ver la escena.
- La variable de estado viewport guarda la información de qué parte de la pantalla vamos a utilizar para montar la escena (desplazamiento respecto a la esquina superior izquierda y tamaño de la pantalla en píxeles).

CREACIÓN DE UN PROGRAMA HOPENGL

- Para definir 3D usamos la función frustum, en la cual definimos 6 valores:



5

EJEMPLOS

EJEMPLOS

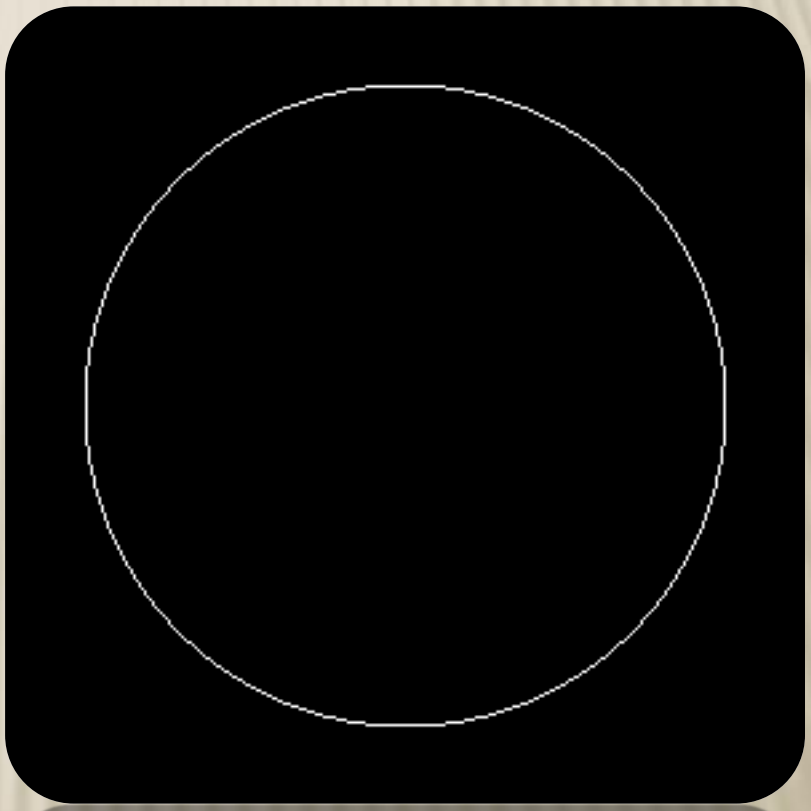
```
import Graphics.UI.GLUT
import Graphics.Rendering.OpenGL

import PointsForRendering

printTea = renderInWindow display

display = do
  clear [ColorBuffer]
  renderObject Solid$ Teapot 0.6
  flush
```

EJEMPLOS

```
module PointsForRendering where
import Graphics.UI.GLUT
import Graphics.Rendering.OpenGL

renderInWindow displayFunction = do
  (progName,_) <-  getArgsAndInitialize
  createWindow progName
  displayCallback $= displayFunction
  mainLoop

displayPoints points primitiveShape = do
  renderAs primitiveShape points
  flush

renderAs figure ps = renderPrimitive figure$makeVertexes ps

makeVertexes :: [(GLfloat, GLfloat, GLfloat)] -> IO ()
makeVertexes = mapM_ (\(x,y,z)->vertex$Vertex3 x y z)
```

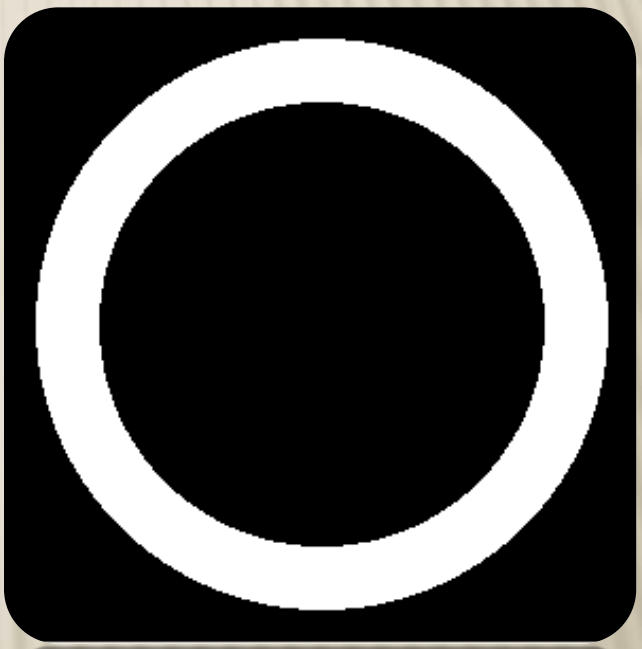
EJEMPLOS

```
module Circle where
import Graphics.UI.GLUT
import Graphics.Rendering.OpenGL
import PointsForRendering

circlePoints radius number
 = [let alpha = twoPi * i /number
    in  (radius*(sin (alpha)) ,radius * (cos (alpha)),0)
   |i <- [1,2..number]]
  where
    twoPi = 2*pi

circle radius = circlePoints radius 100

renderCircleApprox r n
 = displayPoints (circlePoints r n) LineLoop

renderCircle r = displayPoints (circle r) LineLoop
fillCircle r = displayPoints (circle r) Polygon

printCircle = renderInWindow $ do
        clear [ColorBuffer]
        renderCircle 0.8
```



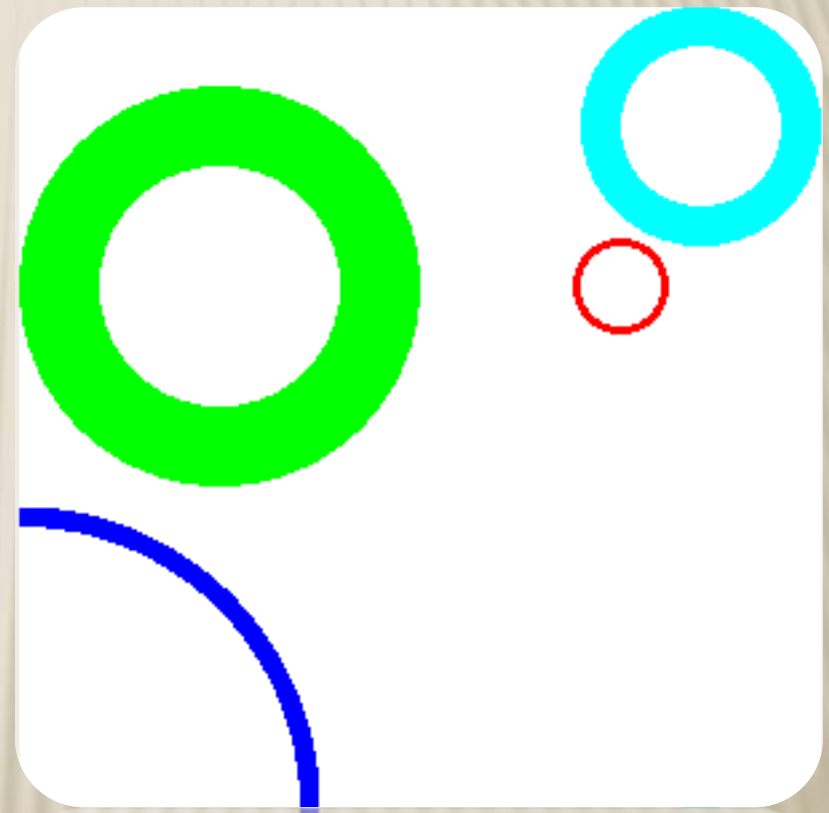
EJEMPLOS

```
module Ring where

import PointsForRendering
import Circle
import Graphics.Rendering.OpenGL

ringPoints innerRadius outerRadius
 = concat$map (\(x,y)->[x,y]) (points++[p])
  where
    innerPoints = circle innerRadius
    outerPoints = circle outerRadius
    points@(p:_) = zip innerPoints outerPoints

ring innerRadius outerRadius
 = displayPoints (ringPoints innerRadius outerRadius)  QuadStrip

printRing = renderInWindow $  do
        clear [ColorBuffer]
        ring 0.7 0.9
```



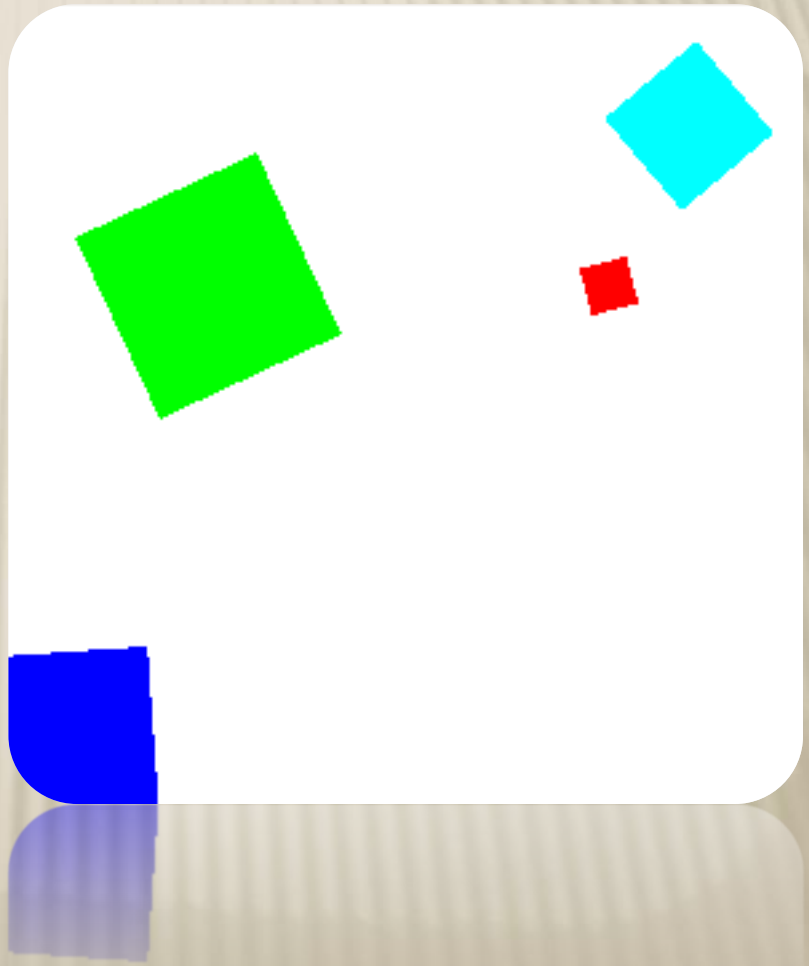
EJEMPLOS

```
import PointsForRendering
import Ring
import Graphics.Rendering.OpenGL

ringAt x y innerRadius outerRadius = do
  translate$Vector3 x y (0::GLfloat)
  ring innerRadius outerRadius

printSomeRings = do
  renderInWindow  someRings

someRings = do
  clearColor $= Color4 1 1 1 1
  clear [ColorBuffer]

  loadIdentity
  currentColor $= Color4 1 0 0 1
  ringAt 0.5 0.3 0.1 0.12

  loadIdentity
  currentColor $= Color4 0 1 0 1
  ringAt (-0.5) 0.3 0.3 0.5

  loadIdentity
  currentColor $= Color4 0 0 1 1
  ringAt (-1) (-1) 0.7 0.75

  loadIdentity
  currentColor $= Color4 0 1 1 1
  ringAt 0.7 0.7 0.2 0.3
```



EJEMPLOS

```
import PointsForRendering
import Graphics.Rendering.OpenGL

myRect width height =
  displayPoints [(w,h,0),(w,-h,0),(-w,-h,0),(-w,h,0)] Quads
   where
     w = width/2
     h = height/2

square width = myRect width width

rotatedSquare alpha width  = do
  rotate alpha $Vector3 0 0 (1::GLfloat)
  square width

displayAt x y displayMe = do
  translate$Vector3 x y (0::GLfloat)
  displayMe
  loadIdentity

printSomeSquares = do
  renderInWindow  someSquares

someSquares = do
  clearColor $= Color4 1 1 1 1
  clear [ColorBuffer]

  currentColor $= Color4 1 0 0 1
  displayAt 0.5 0.3$rotatedSquare  15 0.12

  currentColor $= Color4 0 1 0 1
  displayAt (-0.5) 0.3$rotatedSquare 25 0.5

  currentColor $= Color4 0 0 1 1
  displayAt (-1) (-1)$rotatedSquare 4 0.75

  currentColor $= Color4 0 1 1 1
  displayAt 0.7 0.7$rotatedSquare 40 0.3
```

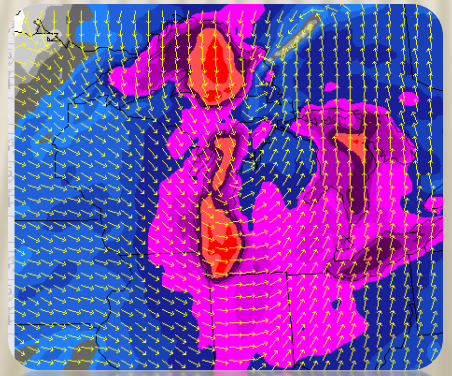


6

UTILIZACIÓN ACTUAL

UTILIZACIÓN ACTUAL

- PolyFunViz: herramienta para visualizaciones científicas (superficies, corrientes, contornos, volúmenes).
- Creado en la universidad de Leeds.



6.- UTILIZACIÓN ACTUAL

- Monadius: clon del Gradius, realizado para conmemorar el 20 aniversario del juego.
- Fue creado por Hideyuki Tanaka y Takayuki Muranushi

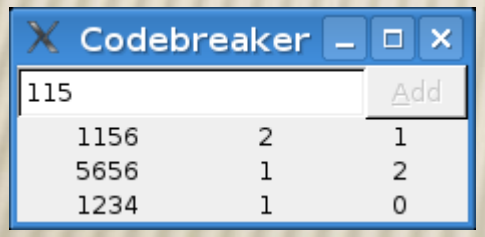
6.- UTILIZACIÓN ACTUAL

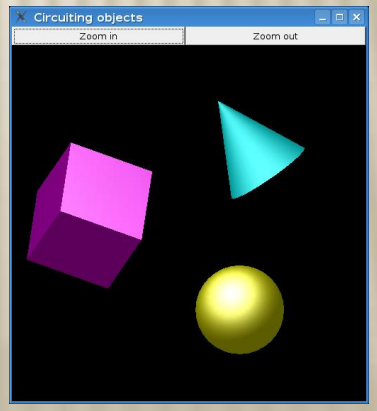
- Frag: juego de disparos en 3ª persona hecho por Mun Hon Cheong como parte de su tesis Functional Programming and 3D Games

- Creado en 2005 en la universidad de New South Wales.

6.- UTILIZACIÓN ACTUAL

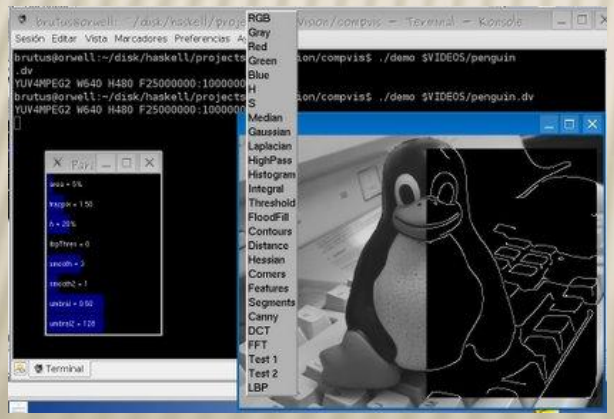
- GrapeFruit: es una librería para FRP (Functional Reactive Programming) orientada a interfaces de usuario. Permite crear sistemas reactivos e interactivos en un estilo declarativo.
- Creada en 2007 por Wolfgang Jeltsch

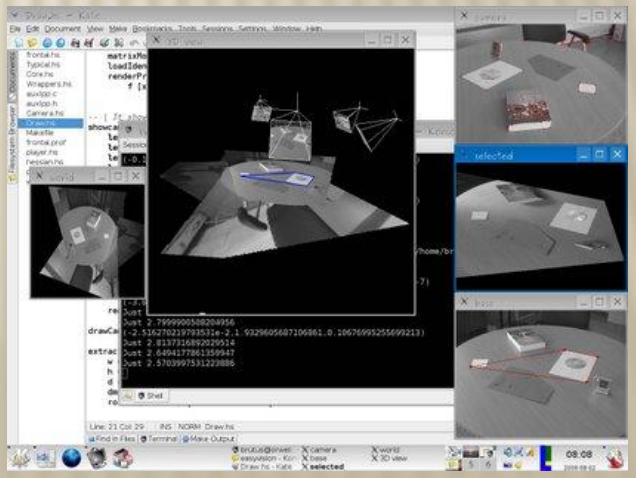




6.- UTILIZACIÓN ACTUAL

- easyVision: sistema experimental en Haskell para procesamiento de imágenes y visión por computador.
- Creado y utilizado en la universidad de Murcia por Alberto Ruiz García.





7

REFERENCIAS Y BIBLIOGRAFÍA

REFERENCIAS Y BIBLIOGRAFÍA

- Referencias generales
  - http://haskell.org/HOpenGL/
  - http://haskell.org/ghc/
  - http://www.cin.ufpe.br/~haskell/hopengl/overview.html (2001) Tutorial de Andrew B.W. Furtado
  - http://guia.ofertaformativa.com/categoria.asp?/Computers/Programming/Graphics/Libraries/OpenGL/Platform_and_OS_Implementations/Language_Bindings/ (2009) Referencia al lenguaje Bindings
  - http://www.taringa.net/posts/info/836225/%C2%BFQue-es-DirectX-y-que-es-OpenGL.html (2009) Guía sobre OpenGL
  - http://lsi.uniovi.es/~labra/FTP/IntHaskell98.pdf (1998) Introducción al Lenguaje Haskell
  - http://markmail.org/message/huqslsdpfvawm3gn (2002-2009) //Andrew y Sven
  - http://www.enciclopediaespana.com/OpenGL.html //OpenGL
- Utilización en la universidad
  - https://www.usc.es/es/centros/enfermaria_stgo/materia.jsp?materia=30026&ano=59&idioma=2
  - http://taee2008.unizar.es/papers/p52.pdf
  - http://www.lcc.uma.es/LCC?-f=LCCProyectos/Proyecto.lcc&-l=spanish&pfc.idpfc=604(2006)
- Enlaces interesantes
  - http://hopl.murdoch.edu.au/findlanguages2.prx?language=&which=byname