**GPU**

* **Renderizado:** proceso mediante el cual se procesa una imagen en 3Dpara conseguir un resultado en píxeles con el fin de que se pueda ver perfectamente en un ordenador.
* **Culling:** proceso por el cual se descartan las partes de los vectores 3D que no vemos en una pantalla 2D.
* **Clipping:** proceso mediante el cual se seleccionan (y marcan) las partes geométricas que se verán, y también las que no sé verán para que a continuación se pueda llevar a cabo el proceso de Culling.
* **Rasterización:** proceso por el cual una imagen descrita en un formato gráfico vectorial se convierte en un conjunto de píxeles.
* **Taxonomía de Flynn:**
  + *SISD:* Single Instruction, Single Data stream (una instrucción y un flujo de datos). Máquinas secuenciales que no tienen ningún paralelismo.
  + *SIMD:* Single Instruction, Multiple Data stream (única instrucción sobre multiples flujos de datos). Las modernas GPU’s se basan en ideas similares a esta. Se adapta a CUDA.
  + *MISMD:* Multiple Instruction, Multiple Data stream (multiples instrucciones sobre un único flujo de datos). Este tipo de arquitecturas tienen sus aplicaciones en entornos donde se necesita una tolerancia a fallos (repitiendo la misma instrucción sobre un dato para evitar posibles errores de HW).
  + *MIMD:* Multiple Instruction, Multiple Data stream (multiples instrucciones sobre multiples flujos de datos). Los sistemas distribuidos son clasificados generalmente como MIMD, bien usen memoria compartida o distribuida.
* **Compute Capabilities:** para hacer uso óptimo de las GPU’s es necesario conocer diferentes características de la tarjeta. Para ello, NVIDIA utiliza un formato estandarizado para especificar estas características, denominado compute capabilities. La catergorización incluye dos números, donde los cambios en la primera cifra implican cambios de generación, mientras que en la segunda implica una revisión.
* **Compilador de Cuda (nvcc, ptx, ocg, cl y gcc):** el compilador de CUDA (NVCC) utiliza un preprocesador (CUDAFE) que divide la aplicación en la CPU y la GPU, creando un archivo preprocesado C para la entrada de GPU: foo.s (lenguaje ensablador) para GPU y foo.cpp para CPU. La entrada de la GPU es procesada por nuestro compilador Open64, llamado nvopencc, que emite un lenguaje ensamblador llamado PTX. La PTX se pasa entonces a OCG, que asigna los registros y la lista de instrucciones de acuerdo con el chip en particular que se está utilizando y optimizando. Tras esto se obtiene código objeto para la GPU, que más tarde da lugar al ejecutable.
* **Cauce gráfico:** inicialmente, a la GPU le llega la información de la CPU en forma de vértices. El primer tratamiento que reciben estos vértices se realiza en el “vertex shader”. Aquí se realizan transformaciones como la rotación o el movimiento de las figuras. Tras esto se define la parte de esto vértices que se va a ver (clipping), y los vértices se transforman en píxeles mediante el proceso de rasterización. Estas etapas no poseen una carga relevante para la GPU. Donde si se encuentra el principal cuello de botella del chip gráfico es en el siguiente paso: el “pixel shader”. Aquí se realizan las transformaciones referentes a los píxeles en la caché, se aplican algunos efectos como el antialising, blending y el efecto niebla.
* **Tipos de memoria:** en función de la latencia de acceso:
  + *Memoria Global:* accesible en R/W por todos los hilos y por la CPU.
  + *Memoria Local:* parte de la memoria privada de cada hilo (memoria virtual).
  + *Memoria Constante:* es global de sólo lectura, puede cargarse en caché de SM para acelerar las transferencias.
  + *Memoria Compartida:* es de tamaño limitado y accesible en R/W por los hilos de un mismo SM.
  + *Registro:* mismo número para todos los hilos, accesible en R/W de forma privada.
* **Clasificación de Hwang-Briggs:** la clasificación de Hwang-Briggs clasifica los computadores en tres tipos:
  + *Computadores pipeline*
  + *Computadores matriciales*
  + *Sistemas multiprocesador*

**Scala**

* **Case Class:** es una construcción que es de ayuda al escribir estructuras de datos no encapsuladas. Se añade “case” delante de la clase que queremos crear, y el compilador de Scala le añade características sintácticas a la clase. En primer lugar, añade un método con el nombre de la clase para poder construir un nuevo objeto con el nombre de la clase. Otra de las características añadidas es que toda la lista de parámetro de la clase implícitamente lleva el prefijo val, por lo que se mantienen como campos. En tercer lugar, se añaden implementaciones naturales como métodos “toString”, “hashCode” y “equals” a la clase. Y por último añade un método copy a la clase para realizar copias modificadas.
* **Pattern Match:** incluye una secuencia de alternativas, cada una empezando por la palabra case. Cada alternativa incluye un patrón y una o más expresiones, que serán evaluadas si el patrón concuerda. El símbolo => separa los patrones de las expresiones. Una expresión match es evaluada intentando que cada patrón en el orden el que fueron escritos.
* **Objetos Singleton:** son instancias de una única clase. Se utilizan en Scala para reemplazar los miembros de clase estáticos que existen en Java. Se definen sin la utilización de “class” para ello.
* **Traits:** son una unidad fundamental de reutilización de código en Scala. Un trait encapsula definiciones de métodos y campos, que pueden ser utilizados mezclándolo entre clases.
* **Mixin:** cuando un trait se mezcla con una clase. El orden de los mixins es importante, van de derecha a izquierda. Cuando se llama a un método de una clase con mixins, se llama primero al método trait más a la derecha, pero si este método a su vez llama a un super, en este caso invocará al siguiente traita a la izquierda.