[Continued on next page]

(54) Title: SCALABLE MULTI-THREADED MEDIA PROCESSING ARCHITECTURE
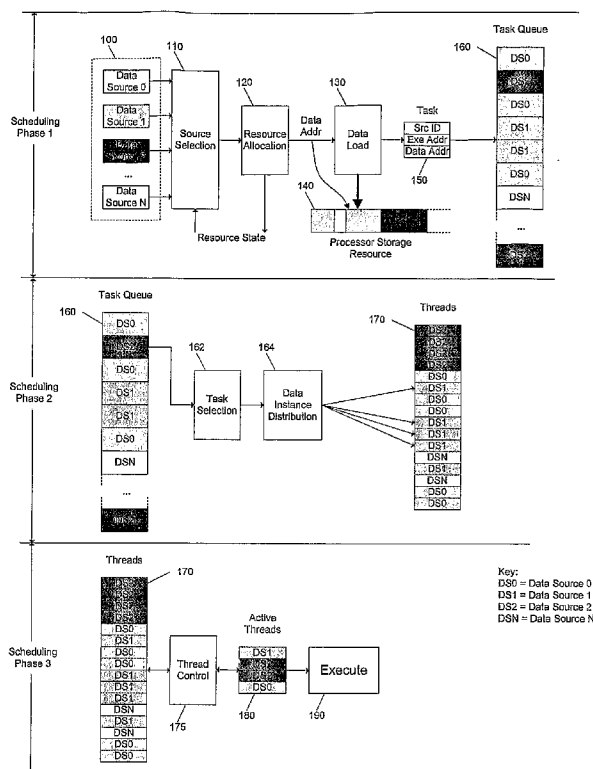
(57) Abstract: A method and apparatus are provided for processing multiple streams of data on a plurality of execution threads. Data is selected from a plurality of data sources (1001). An address in the data storage means (1036) is allocated for the data thus selected. The selected data is then loaded into the allocated address. Following this an execution task comprising the selected data source, the data address and an execution address is constructed and the data task is queued with previously constructed tasks. A determination is made as to which processing resources are required for each task and tasks are selected for execution in dependence on this. Tasks selected for execution are distributed across a plurality of processing threads (170). The allocation of data storage in the data storage means includes the steps of selecting data from one of the data sources and supplying a code execution address to a programmable data sequencer (1004). The code from the code execution address is executed to cause data to be written to the data storage means. Furthermore, a tile based computer graphic rendering system may comprise a plurality of mulit-threaded processor cores. Each processor core is allocated to different sectors of the tile.

Declaration under Rule 4.17:
— *as to applicant's entitlement to apply for and be granted a patent (Rule 4.17(ii))*

Published:
— *without international search report and to be republished upon receipt of that report*

For two-letter codes and other abbreviations, refer to the "Guidance Notes on Codes and Abbreviations" appearing at the beginning of each regular issue of the PCT Gazette.

Scalable Multi-threaded Media Processing Architecture

This invention relates to a system capable of processing multi-media data such video and 3-dimensional computer graphics and in particular to methods of processing multiple parallel streams of such data within a scalable multi-threaded environment.

Background of Invention

International patent application number WO97/38372 (the contents of which are incorporated herein by reference) describes a system for processing multiple real time data sources through a multi-threaded media processing core. The processing core handles a number of execution threads of instructions. A determination is made as to which thread has the highest priority for execution and execution is formed on that thread in accordance with this.

International patent application number WO02/06716 entitled to control a priority lead structure on a multi-threaded processor (the contents of which are incorporated herein by reference) elaborates further on the arbitration schemes which are possible between different executing threads. It provides a method apparatus for controlling execution rates for instruction threads. The rate at which instructions are to be executed is stored and requests are issued to cause instruction to execute in response to the stored rate. The stored rate is reduced in response to execution of instructions and increased in the absence of instructions for execution. Furthermore, instruction rate is controlled by storing the average rate at which each thread should execute instructions. A value representing the number of instructions available not yet issued is monitored and decreased in response to instruction execution. Also, a ranking order can be assigned to a plurality of instruction threads using a plurality of metrics relating to the threads.

In a further development, a technique known as 'Enhanced interleaved multi-threading' works by maintaining a superset of execution threads from which a subset is maintained as active. The number of active threads is determined by the latency of the execution unit to which they are issuing instructions. Threads are then swapped between the active subset and the superset based on resource dependencies, i.e., if an

active thread is waiting for the data to be returned from memory it will be swapped for a currently inactive one that is ready to proceed.

Summary of the Invention

Preferred embodiments of the present invention provide a system which enables a multi-threaded processing pipeline to be used to processes multiple input data streams that represent bulk non real time data sources in a manner that maximises latency absorption and utilisation of pipeline processing resource by using a multi-phased approach to scheduling.

This is performed by using a three phase scheduling system which differentiates the management of data storage within the processing pipeline from the allocation of processing threads within the same pipeline and the scheduling of instructions based on execution resource conflicts.

In accordance with an embodiment of a first aspect of the invention there is provided a method for processing multiple streams of data on a plurality of execution threads comprising the steps of selecting data from at least one of a plurality of data sources in dependence on the availability of storage in a data storage means, allocating an address in the data storage means for the data thus selected, loading selected data at the data address thus allocated, constructing an execution task comprising the selected data source, the data address, and an execution address, queuing the task with other previously constructed tasks, determining which processing resources are required for each task, selecting tasks for execution in dependence on the processing resources available, distributing tasks selected for execution across a plurality of processing threads, determining whether any of the threads will stall because of data dependencies, repeatedly selecting a subset of threads to be active that will not stall due to said data dependencies, and, executing active thraeds.

Preferably an embodiment includes a means for scheduling tasks based on available resource.

Preferably an embodiment includes a means for avoiding prolonged blocking in a data source.

Preferably an embodiment includes a method for loading data into a unified storage area by way of a programmable data sequencer.

Preferably an embodiment may be modified such that the programmable data sequencer is provided on a data processing thread.

Preferably an embodiment provides a method for ensuring data is processed in the correct order.

Preferably the method may be embodied in a tile based rendering system.

Preferably the programmable data sequencer is used for texture coordinate iteration and texture fetches through the system.

Preferably embodiments of the invention are scalable by the different multi processors cores (wherein each multi processor core is matched to a different sector of a tile, the tile size may be adjusted based on the number of attached pipelines, the vertex data is distributed evenly across the pipelines using a pre-scheduling block, and a vent processor is distributed evenly across the pipelines also by a pre-scheduling block).

In accordance with an embodiment of a further aspect of the invention there is provided, a method for allocating data storage in a multithreaded data processing system comprising the steps of selecting data from one of a plurality of data sources in dependence on the availability of data storage, supplying a code execution address to a programmable data sequencer, executing code from the code execution address to cause data to be written to the data storage means.

Brief Description of the Drawings

Preferred embodiments of the invention will now be described in detail by way of example with reference to the accompanying drawings in which:

Figure 1 - illustrates the proposed scheduling phases of an embodiment of the invention discussed above;

Figure2 - illustrates the Improved Multi-Media Processing Architecture embodying the invention;

Figure 3 - detailed diagram of architecture of Figure 2;

Figure 4 - illustrates thread States within Thread Manager of Figure 3;

Figure 5 - illustrates a typical Tile Based Rendering System;

Figure 6 – illustrates a typical HSR unit within a tile based rendering system which may use an embodiment of the invention;

Figure 7 – illustrates a tile based rendering system implemented using an embodiment of the invention;

Figure 8 - illustrates tile sector to Pixel Data Master (PDM) mapping;

Figure 9 – illustrates a system scaled to two pipelines; .

Detailed Description of Preferred Embodiment

Figure 1 illustrates the scheduling phases with the system. The first phase starts with data source selection at 110 of an input from available sources 100 on the basis of there being sufficient resource available in the Processor Storage Resource 140. Allocation of the Processor Storage Resource is then performed at 120 yielding a Data Address within the storage. This address is then used at 130 to load data required for the servicing of the selected source. A task is then constructed at 150 which is composed of (but not limited to) the originating data source ID, an execution address supplied by the data source 100 and the data address. The generated task is then inserted into a task queue 160.

The second scheduling phase involves each task within the queue generated by the first phase being distributed across one or more processing threads within a multi-threaded processing pipeline based on the availability of processing resource within that pipeline. The processing resources required to execute any given task are used to dictate which task within the task queue may be run at 162, specifically the required number of processing threads and temporary storage required to execute those threads. The selected task is then distributed across execution threads 170 at 164.

The third and final execution phase determines which threads may be run at 175 without incurring a stall due to resources that are not currently available e.g. data returned by an external memory read or branch condition that is not currently valid. This is done in a manner that combines the simple resource checking described in patent WO 907/38372 and the technique of "Enchanced interleaved multi-threading". This results in a maintained set of active threads 180, the number of which is tied to the execution latency of "Execute" 190 (typically an ALU). It should be noted that the third phase executes threads to completion at which point both the threads and any allocated memory resource are returned to the system.

4

Figure 2 illustrates the proposed processing architecture to implement figure 1

at a high level. The data sources 810 will from now on be referred as "Data Masters"

on the basis that they are responsible for instigating all processing within the system.

A Coarse Grain Scheduler (CGS) 820 coupled to the data masters undertakes the

activities described above within scheduling phase one. The CGS passes tasks to a

Multi-Threaded Media Processor (MTMP) 830 which implements scheduling phases

two and three described above. Processed data is "emitted" from the MTMP to the

data processing pipelines 840. This system differs to that described in International

patent no. WO97/38372 by the addition of the CGS and the internal differences within

the MTMP associated with the 3 phased scheduling scheme.

Figure 3 illustrates the details of the basic system in greater detail. The data

masters 1001 issue data service requests to a Data Master Selection unit (DMS) 1006

in the CGS 1000 . These data requests are in the form of a count of data instances plus

the size of an individual data instance. A data instance is defined as a unique quanta

of data that may be processed without reference to any other data within the incoming

data stream, and as such may be processed in parallel with all other data instances

supplied by this or any other data master.

The DMS cycles through the data masters looking for one that may be

serviced within the resources that are currently indicated as free by a Resource

Manager 1002 as described in scheduling phase 1 above. When selecting a data

master the DMS also takes into account a minimum data instance count (Referred to

as the Data Master Minimum Service Count) that is indicated by the data masters.

There must be sufficient resource available to service a multiple of the minimum data

instance count stipulated. Once a data master has been selected the DMS allocates the

required resources updating the Resource manager as appropriate before passing the

selected data master's request onto a Programmable Data Sequencer (PDS) 1004. It

should be noted that the resource allocation is in the form of input and output data

storage within the MTMP's shared data store 1036.

Input buffer storage is allocated on a per service request basis, however output

buffer storage needs additional management in certain circumstances. Specifically

certain data types require output buffering to be persistent across multiple service

requests (e.g. pixel data), in these cases the output buffering is managed in terms of

"macro" tasks. A macro task being defined as a group of tasks over which output

buffering must be persistent. To handle this the DMS/resource manager allocates

output buffer storage only on the first task for a data master where there is no output buffer storage currently allocated. This allocation then remains persistent until the data master signals that the macro task is complete, where it may be freed.

It will be noted that it is possible for a data master that requires a large amount of resource for its minimum serviceable quanta to be blocked from processing. This is prevented by the DMS maintaining a timeout counter for each data master. This timeout counter is incremented each time it has a service request rejected, when this counter reaches a pre-determined value the data master concerned will be serviced irrespective of currently available resources i.e. the DMS will be prevented from processing until sufficient resources are available. In addition a software control is made available to control and disable/enable the servicing of each data master. This software control can be used to manipulate the amount of processing time available to each data master.

The PDS is a fully programmable processor that is supplied with a code execution address by a data master selected by the DMS. The indicated code is executed once for each data instance that is being serviced from the data master. The code may use data supplied by the data master to generate data directly into the shared data store 1036 or to fetch data from memory via a DMA unit 1008 into the shared data store 1036. It should be noted that the PDS could be implemented as a thread within the MTMP. However this would then take processing resource from other activities.

Once all data instances from a selected data master have been serviced by the PDS a task is issued to the MTMP's task queue 1022. The execution address for the task is preferably supplied directly from the code executed within the PDS. Thus the PDS operation is equivalent to the "Data Load" step 130 from figure 1.

The task queue is scanned by a Task Control Unit (TCU) 1024 for tasks that can be issued to a Thread Manager 1026, this is any task that,

- has no outstanding execution dependencies,
- whose temporary storage allocation and number of execution threads for the specified minimum number of data instances (same number as Data Master Minimum Service Count) OR
- whose age has reached or exceeded the maximum allowable

Execution dependencies are used to block the execution of a task while there is some external or internal activity that must complete before the task can be allowed to proceed. Examples of such dependencies are (but not limited to) an "external" dependency or a "sequential" dependency.

An external dependency is a condition which is sourced from a piece of hardware external to the MTMP which blocks the task's execution until the condition is released. For example a task may require one of the processing pipelines 1040 to complete before proceeding.

A sequential dependency is applied where a task may update an output buffer register that is also updated on a task that preceded its arrival in the queue. When the task manager sees a task with a sequential dependency set it will allow all tasks from the same data master in the queue that were supplied before it to complete before allowing it to proceed. It should be noted that only tasks from the same data master are blocked in these circumstances, this is why the data master/source ID is included within the task data. In a further modification to this mechanism each task could carry a mask of output registers that are modified, the sequential dependency then being applied selectively only where those masks overlap.

In order to prevent a task with large resource requirements from being blocked for long periods of time each task in the queue has an "Age" counter which is incremented each time a task is rejected for processing by the TCU. When the age counter reaches a defined limit a task that has no outstanding execution dependencies will be selected for processing irrespective of available thread and temporary storage availability.

Once a task has been selected for service its data instances are distributed to threads as they and the required temporary register resources become available as shown schematically in the scheduling phase 2 in figure 1. Temporary register storage is also allocated from the shared data store 1036 with the store being pre divided in a fixed manner between input, output and temporary storage areas which are then managed independently. It should be noted that input and temporary storage be combined into a single partition and both allocated in phase 1 of the scheduling, however this has the disadvantage of reserving temporary storage for longer than necessary.

The Thread Manager 1026 performs the actions required to implement scheduling phase 3 as shown in figure 1. The thread manager maintains a number of simultaneous execution threads of which a subset are active at any given time. Execution of code for the active threads is by round robin e.g. if there are two active threads within the system, instruction fetches are issued for thread 0, threat 1, thread 0, thread 1 etc. As ALU pipelines do not typically produce a result in the same clock cycle as issuing an operation down the pipeline instructions referencing the result of a previous instruction would typically stall. However as instructions are interleaved as described above from threads operating on unrelated data instances the time between potentially dependent register accesses for code running within any given execution thread is increased buy the number of active thread, as such the number of active threads is chosen such that it equals the latency of the ALU pipeline.

Figure 4 illustrates the four possible states of a thread and the activities that cause a transition between those states. Threads start in a 'free' state at 1200 until the TCU 1024 (figure 3) issues data instances to run on them as previously described causing a transition to the 'Ready' state 1210. Threads remain in a ready state until the thread manager places a currently active thread into an inactive or free state, causing a ready thread to transition to 'Active' at 1220. Threads remain active until such point as they encounter a dependency that would result in a stall if they where to continue execution. When a thread hits such a dependency the thread manager moves its state to 'Inactive' at 1230. It should be noted that if no other threads are in a 'Ready' state at this point the thread manager will signal the Instruction Fetch and Decode unit (IFD) 1028 of Figure 3 to insert no-ops into the instruction stream being fed to an ALU 1034 between the IFD and the SDS. A thread remains in an 'Inactive' state until all dependencies that caused its deactivation have been satisfied at which point the thread transitions back to a 'Ready' state at 1210. In order to avoid active threads from blocking execution of any 'Ready' threads for prolonged periods of time the thread manager will change an active thread back to a ready state if a thread exceeds a pre-defined active time limit. At the point an instruction is encountered that is marked as 'Last' (i.e., the last instruction) the thread manager will revert the state of the thread back to 'Free' at 1200. Execution of an instruction marked as "last" also results in any allocated temporary or input storage being freed, although not output storage. The possible dependencies that result in the de-activation of a thread are,

- Code cache miss

8

- Branch dependent on a condition code that is currently invalid
- Data from an external memory read not yet returned
- Attempt to aquire a semaphore that is currently owned by another thread
- Waiting synchronisation

In order to minimise latency for selection of active threads the thread manager updates the status of inactive threads to 'ready' as soon as any outstanding dependencies complete.

The thread manager issues instruction fetches for active threads to the Instruction Fetch and Decode unit (IFD) 1028 which issues fetches to the instruction cache 1030. If an issued fetch causes a cache miss the IFD informs the thread manager of this, which will deactivate the thread as described above, until such time as the Instruction cache retrieves the required cache line. Fetched instructions are decoded and any branch or control instruction executed within the IFD. Any control or branch instruction that references a control resource (such as a condition code) that is not currently valid will result in the IFD informing the thread manager that the thread should be deactivated until such point as the control resource becomes valid. All other instructions are passed to the ALU 1034 for execution. It will be noted that the ALU operates directly on the data contained within the shared data store.

Prior to completion most threads will signal an emit to the Processing Pipelines 1040 which will read required data from the output buffers allocated within the shared data store 1036. When this process is complete the processing pipeline signals that the output buffer resource has been freed.

In a preferred embodiment of the invention the system is adapted for use within a tile base rendering system.

Tile based rendering systems are known. These break down an image to be rendered into a plurality of rectangular blocks or tiles. The way in which this is done and the subsequent texturing and shading performed is shown schematically in figure 5. A primitive/command fetch unit 1501 fetches a stream of commands and primitive data from memory and passes it to a geometry processing unit 1502 which transforms it into screen space using a well known method. The data is then supplied to a tiling unit 1503, which inserts the screen space geometry into lists for a set of defined

rectangular regions, or tiles, 1504. Each list contains primitives that exist wholly or partially in a sub-region of a screen (i.e. a tile). A list exists for every tile on the screen, although it should be borne in mind that some lists may have no data in them.

The tiled data lists are fetch by a tile parameter fetch unit 1505 which feeds them tile by tile to a hidden surface removal unit (HSR) 1506 and from there to a texturing and shading unit (TSU) 1508. The HSR unit processes each primitive in the tile and passes only data for visible pixels to the TSU as described below.

The texturing and shading unit takes the data from the HSR unit and uses it to fetch textures and apply shading to each pixel within a visible object using techniques that are well known.

The TSU then feeds the textured and shaded data to the alpha test unit 1509, fogging unit 1510 and alpha blending unit 1511 which perform alpha test, fogging and alpha blending respectively in a well known manner. Alpha blending is performed to an on chip tile buffer 1512 thereby eliminating external memory bandwidth associated with this operation. On completion of each tile the Pixel processing unit 1514 performs any backend processes such as packing or anti-alias filtering before writing the resulting data out to external memory used to store the rendered scene 1516.

Figure 6 describes the HSR unit of a typical tile based rendering device in more detail. Specifically the rasteriser 1602 tests the depths of incoming primitive pixels against the on chip tile buffer 1608, if a pixel location is found to be visible a "tag" uniquely identifying that object is written to the tag buffer 1604. The contents of the tag buffer need to be passed to the texturing and shading unit any time a pixel requires overwriting with a pixel from an object that is transparent. When this occurs the current contents of the tag buffer are passed to the Sorter 1606, which sorts and passes the contents of the tag buffer to the texturing and shading unit based on object tags such that the texturing and shading unit sees all the pixels associated with each visible object within the tile in one go. This process is referred to as pass "spawning".

Figure 7 describes the further embodiment of the invention as applied to tile based rendering. For reasons of clarity the MTMP control and execution pipe 1852 encompasses the task queue 1022, task control unit 1024, thread manager 1026, thread PC 1032, instruction fetch and decode 1028 and the ALU 1034. It should also be noted that code 1030 and data 1010 caches have also been removed for reasons of clarity only.

In this example the MTMP replaces the function of geometry processing 1502, texturing and shading 1508, alpha test 1509, fogging 1510 and alpha blend 1511 unit from figure 5. In addition the accumulation buffer is replaced with output buffer space allocated from the shared data store. This integration of functionality into a single unit with general programmability results in an exceptionally capable system that may either directly emulate or replace fixed function blocks or replace them with an arbitrary piece of code supplied by an attached application.

The Data masters 1820 are now allocated specific functions as follows. The pixel data master (PDM) 1826 incorporates the functionality of tiled parameter fetch 1505 and hidden surface removal 1506 in figure 5. The PDM feeds pixel processing requests to the DMS 1806 in blocks of N pixels, where in this example N lies between 4 and 128, with 4 being the required minimum due to the well known manner in which pixel processing pipelines must implement rate of change calculations. Each data service request represents multiple pixels in order to minimise the overheard within the PDS 1804 when processing pixel data. It should be noted that the PDM also passes parameters required for triangle rasterisation to the coordinate iterators 1810.

The vertex data master (VDM) 1822 fetches incoming primitives from external memory and passes indices for vertices that should be fetched in each data service request. It should be noted that the VDM also directly passes sideband information to the tiling engine 1862 to enable it to correctly re-construct primitives from the processed vertices.

The event data master (EDM) 1824 is fed with events from the other data masters within the system and from the Host allowing the architecture to be used for processing system events and general commands avoiding the need to interrupt the host processor to service these events.

It should be noted that additional data masters can be added to the system, for example a data master could be implemented that takes a feed from the tiled parameter fetch unit allowing the fetched primitive data to be pre-processed before it is passed to the HSR unit (which would in effect become another processing pipeline). This pre-processing would allow for generation of procedural geometry such as high order surfaces while the tiled lists are being processed allowing for a significant reduction in consumed memory for tiled primitive lists.

The processing pipelines 1860 are also now allocated specific functions with the tiling engine 1862, the pixel processing 1864 and the texturing unit 1866 being directly equivalent to same units in figure 5.

The approach to data processing is as described previously with a few additional functions available. Specifically the PDS may now, for pixel data, also load data into the shared data store by way of a set of coordinate iterators 1810 or the texturing unit 1866.

Pixel processing occurs whenever the PDM's HSR unit spawns a pass which will cause it to signal the DMS that it has data to be processed. When selected the code executed on the PDS loads coordinate and texture data into the allocated area of the shared data store. Data instances within the system represent a single pixel with executed code storing the results into the allocated output buffer within the shared data store. It should be noted that each pixel data instance does not perform an emit to the pixel processing unit 1864 as the output buffer is not finalised until the entire tile has been processed. In order for the tile data to be emitted the EDM 1824 is fed by a signal from the PDM that indicates that the tile is complete. The EDM then generates a task that is used to emit the tiled data to the pixel processing unit. It should be noted that the EDM can be used to generate events in this manner for many other purposes, for example a task could be emitted between passes spawned by the HSR unit allowing processing to be performed on the contents of the tile buffer between each pass. It should be noted that it is possible to perform any amount or type of processing on these events, for example when emitting the completed tile additional processing could be performed to generate levels for a mip-map (a well known technique) for the emitted data.

Vertex processing operates in a similar manner to a generic data source as described previously with an emit being performed to the tiling engine per processed vertex.

In a further extension to the embodiment the architecture may also be scaled. For pixel processing this is accomplished by laying down multiple media processing cores with each core mapped to a different "sector" of the tile as illustrated in figure 8. It will be noted that the split occurs where visible tags are transferred from the HSR unit's tag buffer 2000 to the tag sorters 2010, 2011, with a tag sorter being dedicated to each PDM 2020, 2021, this maximises the level of parallelism between the two pipelines without the need to split the HSR unit at the front end.

As these different sectors are non overlapping they can inherently be processed in parallel across multiple pipelines. Typically as the number of pipelines are increased the tile size is increased in proportion to the number of attached pipes e.g. if a single pipeline is attached to an HSR unit that works on a 16x16 tile, then two pipelines would be attached to an HSR unit that works on a 32x16 tile and so on. Alternatively the size of the sectors can be reduced allowing the tile size to be kept constant, however this will result in a reduction of efficiency when processing pixel data in each pipeline.

Generally speaking the approach of scaling the tile size to match the number of pipelines can continue until the performance of the HSR unit is saturated i.e. it becomes difficult to scale its performance further. At this point scaling can be continued by laying down multiple parallel HSR units.

For vertex processing scalability can be achieved by distributing the incoming primitive data across the available pipelines. This is typically done using either a simple round robin mechanism or by sending blocks of vertices to the pipeline which is the least busy. The former mechanism is preferred as it simplifies the tiling mechanism later in the pipeline.

Figure 9 illustrates the arrangement for a two pipeline system. It should be noted that each of Pipe 1 2250 and Pipe 2 2251 represent both CGS and MTMP previously described separately in order to aid clarity.

The HSR unit 2200 processes incoming tiled parameter lists as previously described. The pixel data masters 2230 and 2231 are each mapped to a different part of the HSR unit's tag buffer as described above. Each PDM is dedicated to a specific media processing core 2250, 2251. It should be noted that the texturing (2260,2261) and pixel processing (2270, 2271) pipelines are duplicated along with the media processing core. The processed pixel data is passed from the MTMP's to the pixel processing units 2270, 2271 at the completion of a tile as previously described. The pixel processing units are duplicated along with the processing units as they have a non overlapping assignment to pixels stored in memory making them directly scalable along with the rest of the pixel processing pipeline.

The vertex data master 2210 and the event data master 2220 each feed service requests into the pre-scheduler 2240. The pre- scheduler maintains a service queue per attached pipeline for each of data masters (VDM, EDM). The queues are filled in

simple round robin order such that processing requests are distributed evenly across the attached pipelines. Processed vertices are passed from the MTMP to the tiling unit 2280 which re-combines them with primitive data generated by the VDM before applying the tiling process as previously described.

CLAIMS

1.   A method for processing multiple streams of data on a plurality of execution
       threads comprising the steps of:
             selecting data from at least one of a plurality of data sources in
       dependence on the availability of storage in a data storage means;
             allocating an address in the data storage means for the data thus
       selected;
             loading selected data at the data address thus allocated;
             constructing an execution task comprising the selected data source,
       the data address, and an execution address;
             queuing the task with other previously constructed tasks;
             determining which processing resources are required for each task;
       selecting tasks for execution in dependence on the processing resources
       available;
       distributing tasks selected for execution across a plurality of processing
       threads; and
       determining whether any of the threads will stall because of data
       dependencies;
       repeatedly selecting a subset of threads to be active that will not stall due to
       said data dependencies; and,
       executing active threads.

2.   A method according to claim 1 in which each data source comprises a
       plurality of data instances each being a quanta of data that may be processed
       without reference to other data in any of the data sources.

3.   A method according to claim 2 in which each data source provides a minimum
       data instance count to the selecting step.

4.   A method according to claim 3 in which a data source is selected in
       dependence on the minimum data instance count.

5.   A method according to claims 2, 3 or 4 in which the selecting step selects a
       plurality of data instances from a data source and provides them to the
       allocation and loading steps to construct a task.

6.   A method according to any preceding claim in which the step of selecting
      tasks for execution selects tasks in dependence on execution dependencies of
      tasks.

7.   A method according to claim 6 in which the execution dependencies include
      external and internal dependencies.

8.   A method according to claim 6 or 7 including the step of blocking tasks that
      have execution dependencies.

9.   A method according to any preceding claim in which the step of distributing
      tasks comprises distributing data instances from a task to a plurality of
      execution threads.

10.   A method according to any preceding claim including the step of incrementing
      an age counter for a task each time it is rejected for execution.

11.   A method according to claim 10 including the step of executing a task when
      its age counter exceeds a predetermined value.

12.   A method according to any preceding claim for use in a tile based rendering
      system.

13.   A method according to any preceding claim in which the step of loading
      selected data comprises supplying a code execution address to a
      programmable data sequencer (PDS) and executing code from the code
      execution address to cause data to be written to the data storage means.

14.   A method according to claim 13 wherein the execution of the code causes data
      to be written directly from the data source to the data storage means.

15.   A method according to claim 13 or 14 wherein the execution of the code
      causes data to be fetched from another data storage location and written into
      the data storage means.

16.   A method according to claim 13, 14 or 15 wherein the PDS comprises a thread
       on a multireaded processing means.

17.   A method according to any preceding claim including the step of allocating
       input data storage in the data storage means each time a data source is
       selected.

18.   A method according to claim 2 wherein a data source includes a plurality of
       data instances which share the same output buffer, and output data storage is
       allocated on the first of these data instances only, until the data source signals
       that the data instances sharing the same output buffer requirements have been
       exhausted.

19.   A method according to claim 12 wherein the method is executed on a plurality
       of processing cores and including the step of allocating different processing
       cores to different sections of a tile.

20.   A method according to claim 19 including the step of selecting the size of tiles
       in dependence on the number of processor pipelines available from the
       plurality of processing cores.

21.   A method according to claim 20 comprising the step of receiving vertex data
       and distributing this across the processor pipelines available.

22.   A method according to claim 20 or 21 comprising the step of receiving pixel
       data and distributing this across the processor pipelines available.

23.   A method according to claim 13 wherein execution of the code causes texture
       coordinate or other data iterated within an iterator to be written directly to the
       data storage means.

24.   A method according to claim 13 wherein execution of the code causes texture
       data to be sampled, filtered and written to the data storage means.

25.   Apparatus for processing multiple streams of data comprising:

a plurality of data sources;

means for selecting a data source from the plurality of data sources in
dependence on the availability of storage in a data storage means;

means for allocating an address in the data storage means for a selected data
source;

means for loading data for the selected data source into said address within the
storage means;

means for constructing a task to be executed, the task comprising the selected
data source, an execution address and the data address;

a task queue for constructed tasks;

means for selecting tasks for execution in dependence on the availability of
processing resources;

means for distributing selected tasks across a plurality of execution threads;

means for determining whether any threads will stall due to data
dependencies;

means for repeatedly selecting a subset of active threads that will not stall due
to said data dependencies; and

means for executing active threads.

26.   Apparatus according to claim 25 in which each data source comprises a
plurality of data instances each being a quanta of data that may be processed
without reference to other data in any of the data sources.

27.   Apparatus according to claim 26 in which each data source provides a
minimum data instance count to the selecting means.

28.   Apparatus according to claim 27 in which the selecting means selects a data
source in dependence on the minimum data instance count.

29.   Apparatus according to any of claims 26, 27 or 28 in which the selecting
means selects a plurality of data instances from the data source and provides

these to the allocating and loading means for use by the task constructing means.

30.   Apparatus according to any of claims 25 to 29 in which the selecting means selects tasks for execution in dependence on execution dependencies of tasks.

31.   Apparatus according to claim 30 in which the execution dependencies include external and internal dependencies.

32.   Apparatus according to claim 30 or 31 including means for blocking tasks that have execution dependencies.

33.   Apparatus according to any of claims 25 to 32 in which the means for distributing tasks comprises means for distributing data instances from a task to a plurality of execution threads.

34.   Apparatus according to any of claims 25 to 33 including means for incrementing an age counter for a task each time it is rejected for execution.

35.   Apparatus according to claim 34 in which the executing means executes a task when its age counter exceeds a predetermined value.

36.   Apparatus according to any of claims 25 to 35 for use in a tile based rendering system.

37.   Apparatus according to any of claims 25 to 36 comprising means for supplying a code execution addressing to a programmable data sequencer (PDS) and means for executing the code from the code execution address to cause data to be written to the data storage means.

38.   Apparatus according to claim 37 wherein the means for executing the code causes data to be written directly from the data source to the data storage means.

39.   Apparatus according to claim 37 wherein the means for executing the code causes data to be fetched from another data storage location and written into the data storage means.

40.   Apparatus according to claim 37 wherein execution of the code causes texture coordinate or other data iterated within an iteration unit to be written directly to the data storage means.

41.   Apparatus according to claim 37 wherein execution of the code causes texture data to be sampled, filtered and written into the data storage means by a texturing engine.

42.   Apparatus according to claim 37 wherein the PDS comprises a thread in a multithreaded processor.

43.   Apparatus according to any of claims 25 to 42 wherein the means for allocating an address in data storage does so each time a data source is selected.

44.   Apparatus according to claim 37 wherein a data source includes a plurality of data instances which share the same output buffer and the means for allocating data storage allocates output buffer data storage in the first of these data instances only, until the data sources signals that the data instances sharing the same output buffer have been exhausted.

45.   Apparatus according to claim 36 comprising a plurality of multithreaded processor cores, each allocated to different sections of a tile.

46.   Apparatus according to claims 45 wherein the size of tiles selected is dependent on the number of processor pipelines available from the plurality of processor cores.

47.   Apparatus according to any of claims 45 or 46 including means a prescheduler to receive vertex data and distribute this across available processor pipelines.

48.   Apparatus according to any of claims 45, 46 or 47 wherein a pre-scheduler
receivies data and distributes this across available processor pipelines.

49.   A method for allocating data storage in a multithreaded data processing
system comprising the steps of selecting data from one of a plurality of data
sources in dependence on the availability of data storage, supplying a code
execution address to a programmable data sequencer, executing code from the
code execution address to cause data to be written to the data storage means.

50.   A method according to claim 49 wherein the execution of the code
causes data to be written directly from the data source to the data storage
means.

51.   A method according to claim 49 or 50 wherein execution of the code
causes data to be fetched from another data storage location and written into
the data storage means.

52.   A method according to claim 49, 50 or 51 wherein the programmable
data sequencer comprises a thread on the multi threaded processing system.

53.   A method according to any of claims 49-52 wherein input data storage
in the data storage means is allocated each time a data source is selected.

54.   A method according to any of claims 49-53 wherein a data source
includes a plurality of data instances which share the same output buffer
requirements, and output buffer data storage is allocated in the data storage
means on the first of these data instances only, until the data source signals
that the data instances sharing the same output buffer requirements have been
exhausted.

55.   A method according to claim 49 wherein the execution of the code
causes texture coordinate or other data iterated within an iteration to be written
to the data storage means.

56.   A method according to claim 49 wherein the execution of the code causes texture data to be sampled, filtered and written into the data storage means by a texting engine.

57.   A tile based computer graphics rendering system comprising a plurality of multithreaded processor cores, each allocated to different sectors of a tile.

58.   A system according to claim 57 wherein the size of tiles selected is dependent on the number of processor pipelines available from the processor cores.

59.   A system according to claim 58 wherein a pre-scheduler receives vertex data and distributes this vertex data across available processor pipelines.

60.   A system according to claim 58 or 59 wherein a pre-scheduler receives data and distributes this data across available processor pipelines.

61.   A method for processing multiple streams of data on a plurality of execution threads comprising the steps of selecting data from at least one of a plurality of data sources in dependence on the availability of storage in a data storage means, allocating an address in the data storage means for the data thus selected, loading selected data at the data address thus allocated, constructing an execution task comprising the selected data source, the data address, and an execution address, queuing the task with other previously constructed tasks, determining which processing resources are required for each task, selecting tasks for execution in dependence on the processing resources available, distributing tasks selected for execution across a plurality of processing threads.

62.   Apparatus for processing multiple streams of data comprising:
a plurality of data sources, means for selecting a data source from the plurality of data sources in dependence on the availability of storage in a data storage

means, means for allocating an address in the data storage means for a selected

data source, means for loading data for the selected data source into said

address within the storage means, means for constructing a task to be

executed, the task comprising the selected data source, an execution address

and the data address, a task queue for constructed tasks;

means for selecting tasks for execution in dependence on the availability of

processing resources, means for distributing selected tasks across a plurality of

execution threads, means for determining whether any threads will stall due to

data dependencies.

63.   Apparatus for allocating data storage in a multithreaded processing

system comprising means for selecting data from one of a plurality of data

sources in dependence on the availability of data storage, means for supplying

a code execution address to a programmable data sequencer, and means for

executing the code from the code execution address and causing data to be
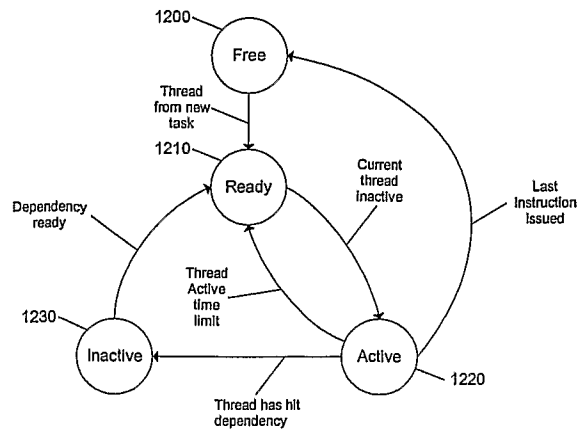
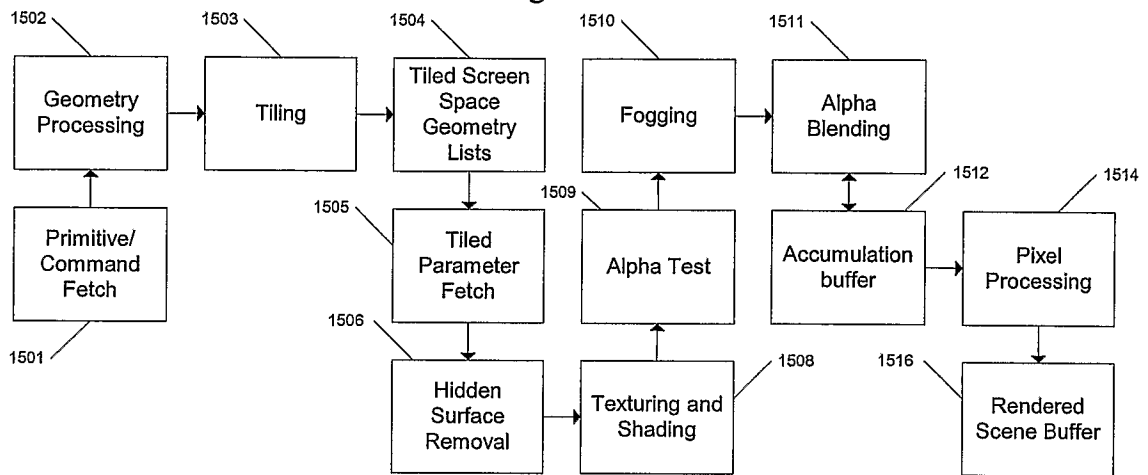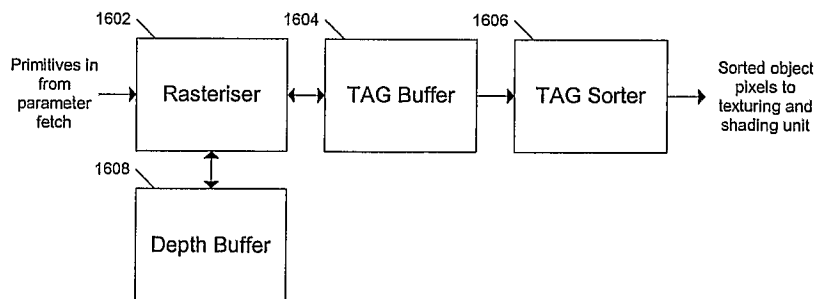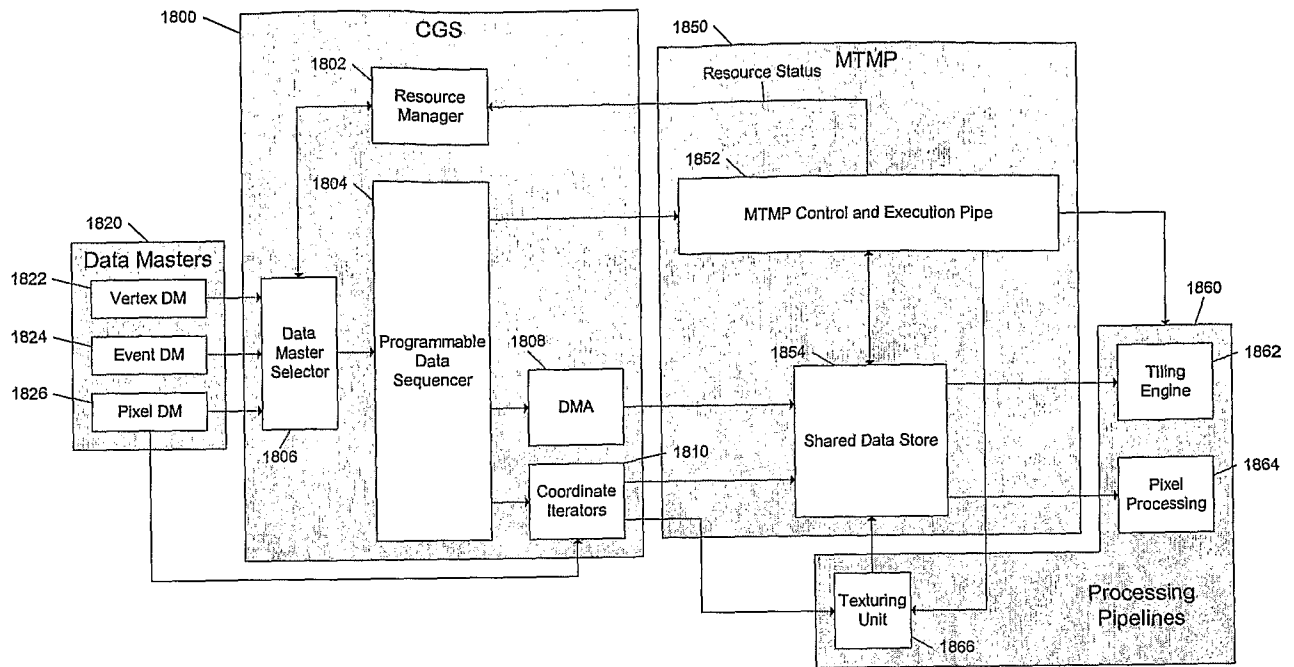written to the data storage.
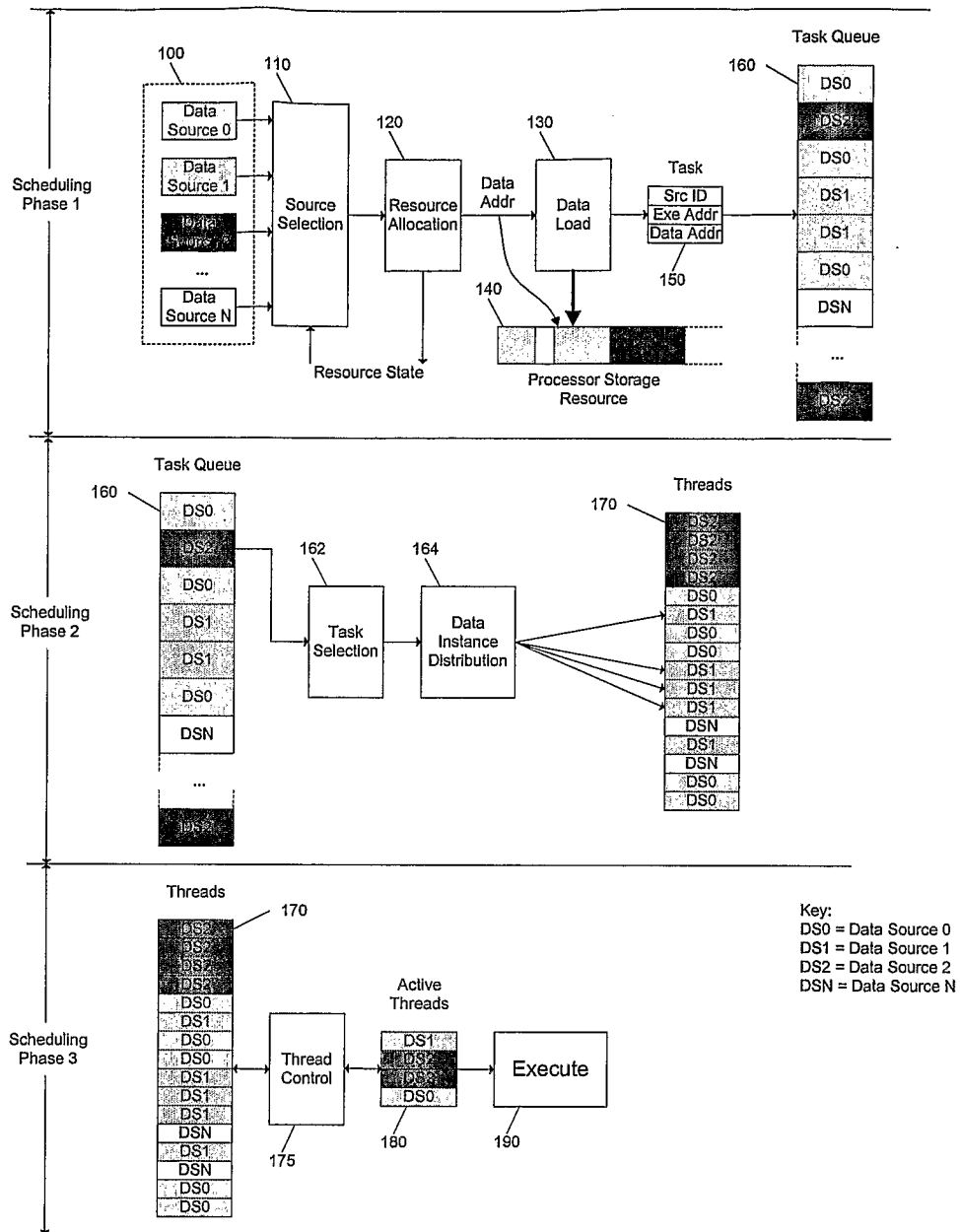
Figure 1
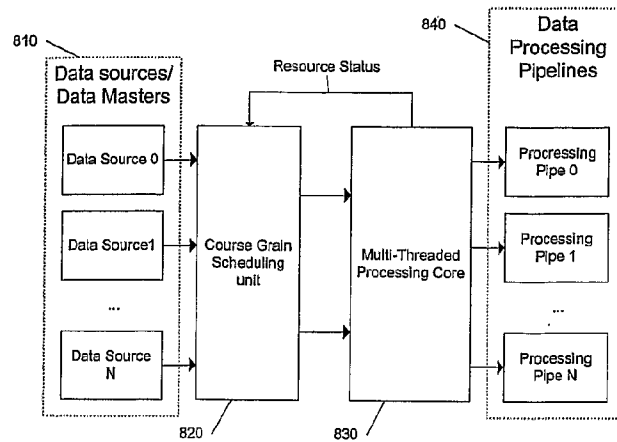
Figure 2
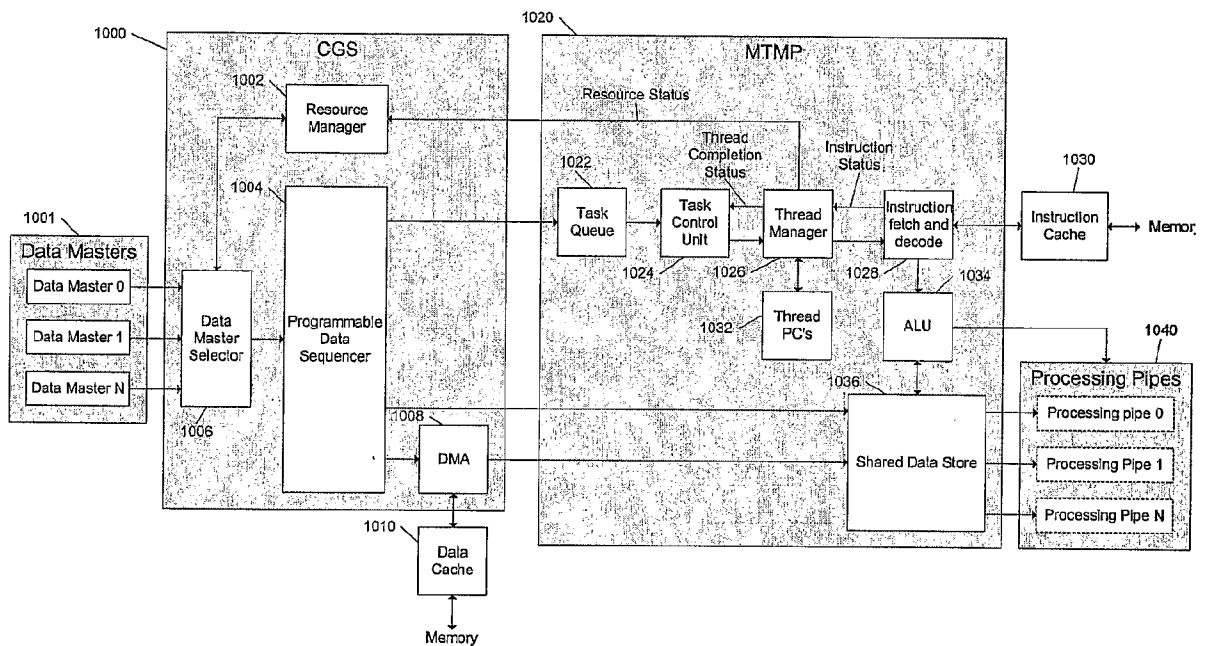
Figure 3

Figure 4

Figure 5

Figure 6

Figure 7

Figure 8

Figure 9